

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



Node.js实战

使用Egg.js+Vue.js+Docker
构建渐进式、可持续集成与交付应用

yugo◎著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>





轻松注册成为博文视点社区用户
(www.broadview.com.cn) ,
扫码直达本书页面。

◆下载资源：本书提供示例代码及资源文件，均可在“下载资源”处下载。

◆提交勘误：您对书中内容的修改意见可在“提交勘误”处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

◆交流互动：在页面下方“读者评论”处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：

<http://www.broadview.com.cn/34912>





内 容 简 介

本书讲解 Node.js 在 Web 开发方面的实际应用, 以一个类 Dribbble 图片画廊应用为实例, 内容包括底层的 Koa.js/Egg.js 框架核心与实现原理, 上层服务的构建、OAuth 服务、JWT 登录认证服务、前后端分离架构, 以及使用 TypeScript 和 Vue.js 实现前后端同构的前端界面, 解决 SEO 问题, 部署与持续集成, 使用时下流行的 Docker 实现 DevOps。最后还介绍了压力测试与上线之后的数据收集的注意事项, 可解决日常企业需求。

本书适合从事 Web 开发并对 Node.js 感兴趣的读者阅读。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目(CIP)数据

Node.js 实战: 使用 Egg.js+Vue.js+Docker 构建渐进式、可持续集成与交付应用 / yugo 著. —北京: 电子工业出版社, 2018.9
ISBN 978-7-121-34912-6

I. ①N… II. ①y… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字 (2018) 第 187921 号

责任编辑: 陈晓猛

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 29.75

字数: 571.2 千字

版 次: 2018 年 9 月第 1 版

印 次: 2018 年 9 月第 1 次印刷

定 价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: 010-51260888-819, faq@phei.com.cn。





前言

为什么要写这本书

前端生态圈的繁荣离不开 Node.js。Node.js 在制作工具方面的表现极其优秀，在开发 Web 方面也有很多历史积累。Node.js 领域的图书很多，比如侧重 Node.js 语法、核心本身，或者侧重调试，而对于 Web 开发，提及 OAuth、JWT 原理的并不多，涉及前端范畴的在线支付、持续集成、Docker 等内容也较少。

笔者比较喜欢体验各种语言，在大学的时候学习和体验了各种语言，包括 C、C++、C#、Java、Python、Ruby、PHP 等，在笔者的网站 nodelover.me 你还会发现有 Go、Rust 的免费视频教程。笔者把大部分精力都花在了 JavaScript 上面，后来才有了这本书。

Node.js 底层还有许多内容笔者也没有弄懂，不过 80% 的业务场景，只需要 20% 的技术能力就可以解决。笔者跟读者一样，都是一个学习者。笔者希望更多的人学习 Node.js、使用 Node.js，使它更加强大。

本书适合的对象

- 有 JavaScript 基础的读者；
- 想要体验完整开发流程的读者；
- 想要精通 Node.js Web 开发的读者；
- 对 Koa.js 和 Egg.js 实现原理感兴趣的读者。

本书也可以作为 Node.js 的入门教程，但是需要你有一定的自学能力，对于一些基础的知识，笔者都会给出视频链接，读者可以自行学习。





本书结构

- 第 1 章：主要叙述了 Node.js 的历史，以及为什么要使用它。
- 第 2 章：讲解 JavaScript 的异步、函数式编程、Koa.js 实现原理，以及 Egg.js 是如何在 Koa.js 上面进行扩展的、Egg.js 是怎样的架构、如何开发出一个 Egg.js 插件并发布到 npmjs。
- 第 3 章：使用 Egg.js 对后端服务进行开发，设计数据库表，构建模型关系映射，建立模型之间的关系。构建安全的 API，使用 JWT 构建登录，使用 OAuth 给第三方开发者开发 API。
- 第 4 章：通过 Vue.js 构建一个简易的后台，通过百行代码实现从后台读取数据库关系，使用 Vue.js 动态地生成对应模型的表单，自动增删改查。
- 第 5 章：使用 TypeScript 与 Vue.js 搭建 SSR 服务端渲染环境，构建友好的 SEO，开发前端显示界面。
- 第 6 章：使用 Docker 部署我们的应用，讲解如何编写 Dockerfile、docker-compose.yml，如何实现通过修改一行代码提交修改，然后自动部署服务。
- 第 7 章：性能分析与优化，包括服务器性能优化、用户追踪、前端性能优化。

勘误与支持

由于部分 npm 组件 API 的变动与 Node.js 的发展，以及笔者的疏忽、水平有限，书中总会有一些不足之处，还望读者批评指正，可以通过以下方式与笔者联系。

GitHub issues: <https://github.com/MiYogurt/nodejs-shizhan>

QQ 群: 325568224

致谢

首先要感谢的是曾经努力的自己，对他说一句“你真棒”。其次感谢父母对我的支持，假如没有父母的支持，可能我就不会有那么多的精力来做这件事情。感谢陈晓猛编辑的耐心指导、审稿、修改，在他的修改下，使得本书有更好的阅读体验。最后感谢的是 Node.js 社区的各位开发者，我们都是站在巨人的肩膀上，感谢巨人们。





目录

第 1 章 Node.js 的优势	1
1.1 为什么是 JavaScript 语言	1
1.2 为什么经常说 Node.js 不适合大型应用	3
第 2 章 Egg.js 框架核心原理与实现	6
2.1 异步基础	6
2.2 Koa.js 基础知识	15
2.2.1 Koa.js 中间件核心代码	16
2.2.2 Koa.js 插件	18
2.3 Egg.js 基础知识	21
2.3.1 实现 egg-core	22
2.3.2 实现 egg-init	26
2.3.3 实现 egg-cluster	30
2.4 Egg.js 插件	33
2.4.1 egg-socket.io	33
2.4.2 原理解读	39
2.5 制作一个 Egg.js 插件	43
第 3 章 构建后端 API 服务	52
3.1 安装相关组件	52
3.2 发布一个插件	59
3.2.1 创建 Flash 插件	59
3.2.2 使用 egg-msg-flash	72
3.2.3 使用 egg-y-validator	73





3.3	规范化	73
3.3.1	添加新的 scripts 支持 ESLint 自修复	74
3.3.2	添加插件支持	74
3.3.3	prettier 格式化工具	76
3.3.4	同步代码编辑器配置	76
3.4	第一个 JSON 请求	77
3.4.1	给全局添加一些方法	77
3.4.2	全局化一些东西	84
3.4.3	自动路由	86
3.4.4	创建 PostMan 测试	88
3.5	注册服务	91
3.5.1	Invitation 模型	91
3.5.2	注释	93
3.5.3	User 模型	96
3.5.4	修改控制器	97
3.5.5	添加验证逻辑	98
3.5.6	帮助方法	99
3.5.7	User 服务	101
3.5.8	PostMan 测试	103
3.6	登录服务	104
3.7	邮件与调试	115
3.7.1	理解发送邮件的原理	115
3.7.2	安装邮件插件	115
3.7.3	环境与调试	116
3.7.4	全局调试	118
3.7.5	VSCode 全局调试	121
3.7.6	发送验证邮件	122
3.7.7	添加逻辑	125
3.7.8	验证	135
3.8	构建 RESTful API	137
3.8.1	什么是 RESTful API	137
3.8.2	创建 RESTController 基础类便于继承	138





3.8.3	测试 Images RESTful API	141
3.8.4	构建后台的 REST 路由	143
3.8.5	构建控制器	145
3.8.6	测试后台路由	148
3.8.7	关于验证	149
3.9	安全地开放 API	151
3.10	实现 OAuth 接口	158
3.10.1	实现授权码官方文档所要求的接口	158
3.10.2	实现刷新验证码接口	167
3.10.3	实现 authenticate 所需接口	169
3.11	完善 OAuth 与测试	170
3.11.1	发放 Token	170
3.11.2	新建客户端项目	172
3.11.3	测试 OAuth	173
3.12	支付宝支付	176
3.12.1	什么是非对称加密	176
3.12.2	注册支付宝	176
3.12.3	生成非对称秘钥	176
3.12.4	实现	177
3.12.5	添加路由	180
3.12.6	内网穿透	180
3.12.7	测试	182
3.13	社会化登录	183
第 4 章	构建后台管理页面	189
4.1	后端开发	189
4.1.1	安装 VSCode 插件	189
4.1.2	获取脚手架	189
4.1.3	安装依赖	190
4.1.4	修改代码	190
4.1.5	跨域请求	191
4.1.6	修改后端代码支持跨域	192
4.1.7	在前端添加存储	198





4.2	模型列表	200
4.3	添加数据	209
4.4	修改逻辑	220
第 5 章 前端界面设计与实现		228
5.1	搭建前端开发环境	228
5.1.1	开始	228
5.1.2	创建 Header 头部	229
5.1.3	将变量提取出来	234
5.1.4	添加路径重写	235
5.2	AppFooter 组件	237
5.2.1	做一些配置	237
5.2.2	创建 src/components/layouts/AppFooter.vue	238
5.2.3	网络识别信息	247
5.2.4	修改一下全局样式	247
5.2.5	查看页面	248
5.2.6	提升编译速度	248
5.3	首页	249
5.4	替换成为真实数据	269
5.4.1	完成后端 Image API	269
5.4.2	修改首页的代码	271
5.4.3	添加 API 逻辑	276
5.4.4	效果	278
5.5	图片详情页	278
5.5.1	创建路由	279
5.5.2	安装依赖	279
5.5.3	创建视图	279
5.5.4	添加插件	285
5.5.5	创建评论组件	286
5.5.6	测试	290
5.5.7	关于服务端访问 DOM	290
5.6	注册页面	294



5.6.1	注册路由	294
5.6.2	新建 signup.vue 页面	295
5.6.3	增强错误提示	299
5.7	登录页面	299
5.8	完善详情与评论	310
5.9	个人中心	321
5.10	创建图片	336
5.10.1	创建又拍云存储	336
5.10.2	添加后端 API	338
5.10.3	前端界面	340
5.10.4	测试	348
5.11	团队	349
5.11.1	功能是如何工作的	350
5.11.2	数据库	350
5.11.3	后端	356
5.11.4	前端	363
5.11.5	测试	372
第 6 章	部署与运维	374
6.1	认识 Docker	374
6.1.1	解决了什么问题	374
6.1.2	使用 Docker 的流程	375
6.1.3	安装 Docker	378
6.1.4	使用加速器	378
6.1.5	下载一个基础镜像	379
6.1.6	hello world	379
6.2	手动构建镜像	380
6.3	编写 Dockerfile 文件	384
6.4	Docker Compose	387
6.4.1	安装 docker-compose	387
6.4.2	命令行接口	388
6.4.3	Egg.js 简单实例	389
6.4.4	增加服务	391

6.5	集群	396
6.5.1	Docker 集群	396
6.5.2	集群初始化	396
6.5.3	实例	397
6.6	持续部署	400
6.6.1	部署主机免密码登录	400
6.6.2	客户端钩子	401
6.6.3	使用服务端钩子进行部署	403
6.6.4	使用 shipit	404
6.6.5	使用 Ansible 部署	406
6.7	持续集成	409
6.8	Kubernetes 集群	423
6.8.1	简单使用	423
6.8.2	如何创建应用	425
6.8.3	命令行管理	430
6.8.4	通过 UI 创建应用	433
6.8.5	添加持续集成	439
6.8.6	固定 IP 地址	441
6.8.7	部署前端	442
第 7 章	性能分析与优化	448
7.1	服务器性能分析与测试	448
7.2	用户追踪	458
7.2.1	百度分析	458
7.2.2	Google 分析	460
7.2.3	其他付费服务	461
7.3	前端性能分析与优化	461
7.3.1	lighthouse	461
7.3.2	sonarwhal	462
7.3.3	图片压缩	464
7.3.4	错误上报	465
7.3.5	接收用户反馈	466

1 chapter

第 1 章

Node.js 的优势

1.1 为什么是 JavaScript 语言

JavaScript 拥有低成本、非常流行和运用范围广三大优势。

1. 低成本

一个网站包括前端界面和后端数据存储，无论选择何种语言来实现后端的逻辑，前端的实现都离不开 HTML、CSS 和 JavaScript 这几种技术。

也就是说，这几种技术是必修课，假如后端的逻辑也可以使用 JavaScript 来写，就可以减少学习一门后端语言的成本。所以只要安装了 Node.js 运行环境，就可以在操作系统中运行 JavaScript，读者只需把 Node.js 当作运行 JavaScript 的一个软件即可。

其实这只是减少成本的一个方面，而且只是表面上减少学习一门语言的成本。想要写好后端的逻辑，一些基础知识同样是必修课，这些学习成本是无法减免的。

另一方面的低成本其实是在包(Package)上面，这里的包指的是其他开发者开发好的代码，这个代码可以提供一些好用的函数、功能等，然后放在 NPM (node package manager) 上，供任何人下载使用。在后面的章节里，笔者会使用大量来自 NPM 优秀的包来提高开发效率。

2. 非常流行

打开 Photoshop CC 的安装目录，会发现 Node.js 的运行环境，即 node 执行程序。在 Linux/UNIX 操作系统中它通常是一个叫 node 的二进制执行程序，在 Windows 中，它通常被称为 node.exe 的可执行程序。

使用 Node.js 的公司并非只有 Adobe 公司，如阿里巴巴、腾讯、Facebook、Google、百度等公司都在使用。

基本上只要跟 Web 有关的公司都会直接或间接地用到 Node.js，Node.js 通常还被用来作为前端的构建工具的一部分——用来压缩代码、减少文件体积。在压缩过程中，避免不了要读取文件内容，这就会用到 Node.js 提供的 fs（File System）文件系统模块。

另外一方面则体现在 Node.js package 的数量上面，从 npmjs.com 上面的数据来看，开源的 package 共计 475 000 个。目前笔者看到的当前 package 周下载量为 3 255 222 628 次，每个月超过 7 百万的开发者使用 npmjs.com 来寻找和分享 package。

3. 运用范围广

自从 Node.js 问世，JavaScript 的生态越来越完善，JavaScript 能做的事情越来越多。

目前 JavaScript 的主战场还是在浏览器端，虽然通过一些框架，例如，Facebook 的 React Native、Apache 的 Weex 等可以使用 JavaScript 编写移动端框架，但是相比较 Swift、Object-C 与 Java、Kotlin 所写的原生应用还是有一定差距的。例如，部分性能问题（需要自己调优）、只能调用封装好的组件（用原生语言封装好供 JavaScript 调用，对于特殊需求，官方并不一定都能满足）等。这些框架只是提供一种解决方案而已，没有任何一种解决方案是完美的。

所以有的初学者总是关注负面，聚焦于性能问题，而忽略了它的优点，其实笔者认为可能初学者都被自己误导了。首先要确认的一点是，很多有经验的人通常工作在一些大型企业里面，而这类企业的用户量其实是非常多的，那么在非常多用户量的情况下，提升一点点性能确实能节约非常多的成本，毕竟积少成多，所以他们才会特别关注性能，而初学者在不明确上下文的情况下，就很容易被带到“沟里”去。

其实在初创公司里最不用关心的就是性能问题，除了代码编写（如内存泄露）导致的问题，只要是合理的架构，就非常容易横向扩展。而且就算是框架的问题也是可以解决的，腾讯大量地将 React Native 应用到 QQ 中，而淘宝则在“双 11”活动中大量实践了 Weex，依然没有出现性能问题。

而桌面端也有 Electron 框架，GitHub 开源的代码编辑器 Atom 其实就是基于 Electron 框架的，后来由于性能问题，部分组件用 C++ 重写了，目前来说界面还是用 HTML、CSS、JavaScript 写的。而开发物联网应用，则有 iot.js，它是三星开发的嵌入式 JavaScript 执行环境。

而服务端，本书用到的 Egg.js 就是一个非常好的例子，对于 Egg.js 的更多知识，在后续的章节都将一一讲解。其实同类型的解决方案还有很多，这里只是抛砖引玉地阐述 JavaScript 在某一方面是有解决方案的。

从上面几点可以看出，JavaScript 是非常适合个人独立开发者、初创公司使用的一门语言。

1.2 为什么经常说 Node.js 不适合大型应用

首先要明确什么是大型应用，其实这是仁者见仁、智者见智的问题，并且它是一个哲学问题，不是一个技术问题。假如有人问你，一个可以进行线上销售的网站，比如优衣库，大不大？你可能会说大，因为这与你平常所见的博客、企业官网等逻辑相比较确实复杂很多。或者说小，那么说明你开发过比它还复杂的系统。那么相比较淘宝而言呢？大和小的对比是要有参照物的。

1. 应用的组成

一个完备的 Web 应用可能只由一门语言或者一种技术构成吗？不可能。因为一个完备的 Web 应用其实是多门技术的综合体，解决某个问题有非常多的解决方案，比如后端的逻辑解决方案就非常多，Java、PHP、Python、Ruby 等都可以。

简单地概述，应用的组成内容可能包括：

- Web 界面显示逻辑；
- 后端业务逻辑；
- 缓存；
- 数据库；
- 消息队列。



其实还可以加入日志分析、数据分析等，只是上面几个最广为人知而已。

2. 应用的种类

- I/O 密集型；
- CPU 密集型。

就常见的互联网产品而言，它的瓶颈并非在后端业务的逻辑上，而是在 I/O 上，即返回给用户看的数据的读入与输出。相对于应用程序而言，读入指的是从数据库里获取数据，而输出指的是将这些数据经过一定的处理输出到用户的浏览器，那么这就是 I/O 密集型。

而 CPU 密集型是指做频繁计算任务的应用，Node.js 在这方面确实是短板。

3. 应用服务的过程

如图 1-1 所示，用户通过浏览器发送请求，由网卡接收 TCP 连接，通知内核，内核再去调用相对应的服务端程序。

Request 请求过程



图 1-1

Response 返回过程

如图 1-2 所示，Web 应用要返回数据，首先要获取数据，通过内核调用磁盘的驱动程序，把数据读入缓存，这样可以在 Web 应用程序中获取数据并进行数据处理，最终调用内核，将数据通过网卡发送给客户端。

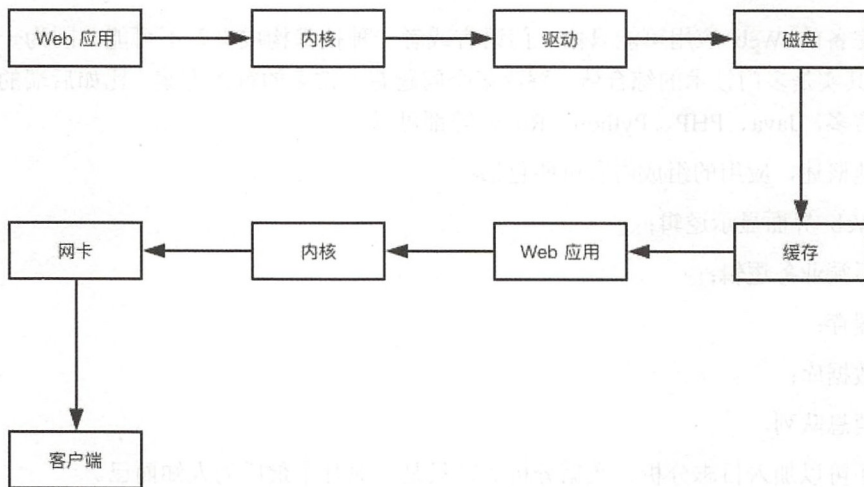


图 1-2

4. 应用的瓶颈

通常 I/O 密集型的瓶颈会在磁盘的读写上，所以在购买云服务器的时候可以购买 SSD 的磁盘来提升性能，一般数据库软件的数据都是存储在文件上面的。首先考虑添加内存型缓存来解决这个瓶颈，缓存经常访问的数据，看能否解决当前场景的问题，比如使用 Redis。其次才考虑搭建或扩充数据库集群来提高并发。

而 CPU 密集型的应用瓶颈则在 CPU 上，只能增加 CPU 处理核心来解决瓶颈。

5. 分布式应用

大型的普通应用与分布式应用其实是不同的概念。读者可以把分布式应用简单地理解为一个团队，每一个成员都是一个节点，一个大的项目要让成员合作完成，那么成员与成员之间就存在一些沟通成本，甚至有的成员与成员之间勾心斗角，说话阳奉阴违、推脱责任，也有可能成员生病在家休养，无法工作，等等。在面对这些问题的时候，Node.js 的优势并不能很好地显现出来（并非不可以做，只是没有完善的基础设施）。

分布式的真正定义是，在多台不同的服务器中部署不同的服务模块，以进程为基本单位，派发到服务器上，通过远程调用（RPC）通信并协同工作，最终对外提供服务。

相比较 Node.js 目前的分布式基础设施，Go 语言的基础设施则完善多了，特别是在 Docker 这个项目上，充分证明了 Go 语言的优势，这也是为什么 Node.js 社区“大牛”TJ Holowaychuk 转向 Go 语言，因为他要开发分布式应用。

其实没必要过分地关心分布式的问题，毕竟 JavaScript 最初只是一个运行在浏览器端的脚本语言而已，JavaScript 不是万能的，为什么一定要把它用在操作系统级别的开发上呢？寻找一个更合适的语言不是更好吗？就像此刻我们选择 JavaScript 构建 Web 应用一样。

6. 多进程的 Node.js

了解了以上的一些知识点，现在读者应该知道，Node.js 跟大型应用关系不大。大多数学习 Node.js 的开发者是前端开发者，所以对后端的基础知识并不了解，在网络上搜寻一些资料的时候发现 Node.js 只能利用单核，而又听说 TJ Holowaychuk 转向 Go 的阵营，所以有的开发者就产生了 Node.js 不适合开发大型应用的疑问。

Node.js 只能利用单核的问题已经被解决了，后面使用的 Egg.js 框架中的 Egg-Cluster 模块就利用多进程非常好地解决了这个问题。



第 2 章

Egg.js 框架核心原理 与实现

2.1 异步基础

本节将介绍什么是异步，并阐述 Node.js 中异步的历史，以及如何进行异步编程。笔者也录制过一些关于异步的教学视频，在 <https://nodelover.me/course/js-async> 上可以免费获取。

1. 什么是异步

我们用一个故事来简单地概述异步。小雷想要给孩子买奶粉，所以去了经常买奶粉的地方，发现店前的公告板上写着本周日不上班。小雷不可能一直等到星期一奶粉店开门，所以小雷只好去其他地方给孩子买个尿布然后回家，等到星期一再来看看。奶粉店关门就是一个耗时任务，让买奶粉这件事没办法立刻完成，只有等待，在奶粉店面前死等就是同步。而且既然出来了，通常还会考虑家里还缺什么，比如说尿布不多了，所以就去其他的地方买个尿布再回家。这里放下买奶粉这件暂时无法完成的事情，而去做可以立即完成的事情，比如买尿布，这就是异步。

2. 回调实现异步

实现异步的本质是回调，我们从 Node.js 的 API 来看一下异步与同步的区别，让读者有一个直观的感受，代码如下：

```
const fs = require("fs");
```



```

let data1 = fs.readFileSync("./mock.txt");

console.log(data1.toString());

let data2 = fs.readFile("./mock.txt", (err, data3) => {
  console.log("data3 ++++");
  console.log(data3.toString());
});

console.log(data2);

```

运行的结果如下，`readFileSync` 是同步的方法，调用之后立刻返回了值，所以可以直接打印出来。而当打印 `readFile` 的返回值的时候，可以发现返回值是 `undefined`，说明根本就没有返回值，而数据是在 `data3` 的回调里面打印的。

```

hello node.js
undefined
data3 ++++
hello node.js

```

像这种通过回调实现异步的情况，可以说在 Node.js 的 API 中比比皆是，这种原始的方式容易被人诟病，当系统变大时，异步是不能直接返回值的，所以一个异步只能跟着另外一个异步，久而久之嵌套越来越多，那么这个代码就没办法维护了，这通常被开发者称为回调地狱。现在不讲如何优化，先继续学习如何创建一个异步的函数。这里通过 `setTimeout` 来模拟异步任务，`setTimeout` 其实也是回调，像这种回调错误不太好处理，所以默认第一个参数为错误，假如有第一个参数，则说明发生了错误，那么就要进行相应的处理。

```

function my_async_function(name, fn) {
  setTimeout(() => {
    fn(null, "-" + name + "-");
  }, 3000);
}

my_async_function("hello node.js", (err, name) => {
  if (err) {

```



```
        console.error(err);
    }
    console.log(name);
  });
```

3. 事件实现异步

事件在 Javascript 和 Node.js 中很常见，在前端的浏览器中最常见的就是页面交互的点击事件，而在后端的 Node.js 环境中，API 也提供了 events 模块，现在实现一个事件类，事件的本质就是发布订阅设计模式。之所以能异步，还是因为回调。

```
class Evente {
  constructor() {
    this.map = {};
  }
  add(name, fn) {
    if (this.map[name]) {
      this.map[name].push(fn);
      return;
    }

    this.map[name] = [fn];
    return;
  }

  emit(name, ...args) {
    this.map[name].forEach(fn => {
      fn(...args);
    });
  }
}

let e = new Evente();
e.add("hello", (err, name) => {
  if (err) {
    console.error(err);
    return;
  }
});
```

```
    console.log(name);
  });

  e.emit("hello", "发生了错误");
  e.emit("hello", null, "hello nodejs");
```

事件模块是学习 Node.js 的基础，Node.js 的不少模块都是基于事件模块构建的，因为事件的里面还是回调，所以对于处理错误，还是把第一个参数作为错误进行传递。

对于事件，通常会有一个名字，比如单击事件等，所以要有一个 `name` 参数，然后把 `name` 参数作为 `key` 放到 `map` 里，当触发 `emit` 方法的时候，获得对应的回调列表进行调用即可。上述的代码其实还可以做一点优化来支持链式调用：

```
class ChainEvent {
  constructor() {
    this.map = {};
  }

  add(name, fn) {
    if (this.map[name]) {
      this.map[name].push(fn);
      return;
    }

    this.map[name] = [fn];
    return this;
  }

  emit(name, ...args) {
    this.map[name].forEach(fn => {
      fn(...args);
    });
    return this;
  }
}

let e2 = new ChainEvent();

e2
```

```
.add("hello", (err, name) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(name);
})
.emit("hello", "发生了错误")
.emit("hello", null, "hello nodejs");
```

把事件加入异步编程里面，只需要在异步的方法里面触发这个事件即可。其实本质上都是回调，只是换了一种形式而已，并没有从根本上解决这个问题，对于简单的异步用事件甚至感觉变麻烦了，代码如下：

```
const fs = require("fs");

// 不用匿名函数
function readFn(err, data) {
  console.log(data.toString());
}
fs.readFile("mock.txt", readFn);

// 事件形式
let e2 = new ChainEvent();
e2.add("readFn", readFn);
fs.readFile("mock.txt", (err, data) => {
  e2.emit("readFn", err, data);
});
```

4. 观察者模式实现异步

```
function create(fn) {
  let ret = false;
  return ({ next, complete, error }) => {
    function nextFn(...args) {
      if (ret) {
        return;
      }
    }
  }
}
```



```
    next(...args);
  }

  function completeFn(...args) {
    complete(...args);
    ret = true;
  }

  function errorFn(...args) {
    error(...args);
  }

  fn({
    next: nextFn,
    complete: completeFn,
    error: errorFn
  });

  return () => (ret = true);
}

let observable = create(observer => {
  setTimeout(() => {
    observer.next(1);
  }, 1000);
  observer.next(2);
  observer.complete(3);
});

const subject = {
  next: value => {
    console.log(value);
  },
  complete: console.log,
  error: console.log
};

let unsubscribe = observable(subject);
```

上述 API 借鉴于 Rx.js, Rx.js 相当于 Promise 的增强版, 这里没有再探究操作符, 结果如下:

2

3

之所以会这样，是因为 `complete` 调用之后，把 `ret` 置为了 `true`，所以在 1 秒后再次调用 `next` 就不生效了。本来 `complete` 的字面意思就是完成，完成之后 `next` 就不应该生效，当有错误的时候用 `error` 发出即可。这里用了高阶函数，即函数返回函数，这里有 3 层函数。为了实现这些设计模式，还需要开发者有一定的编程功底。

5. Promise 异步

Promise 是 ES6 的新特性，专门为了处理异步而生，本质上还是回调，只不过 Promise 成为 JavaScript 的标准，所以我们通常会用 Promise 来实现异步编程。在浏览器端，有的低版本浏览器没有 Promise，那就需要添加 Promise 的兼容实现库，比如 `promise-polyfill`、`babel-polyfill` 等。

```
const getName = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("nodejs");
  }, 50);
});
```

```
const getNumber = Promise.resolve(1);
const getError = Promise.reject("出错啦~");
```

```
getError.catch(console.log);
```

```
Promise.all([getName, getName])
  .then(console.log)
  .catch(console.log);
```

```
Promise.race([getName, getName])
  .then(console.log)
  .catch(console.log);
```

```
getName
  .then(name => {
    console.log(name);
    return 20;
  });
```

```
  })  
  .then(number => {  
    console.log(number);  
  });
```

运行结果如下：

```
出错啦~  
nodejs  
[ 'nodejs', 'nodejs' ]  
nodejs  
20
```

Promise 的 API 并不多，常用的还是 all 方法。all 方法接受一个 Promise 对象的数组，当所有 Promise 都返回的时候才会执行 then 后面的方法，假如有任何一个 Promise 返回错误，则都将失败而调用 catch 里面的回调。all 方法通常用来处理并行关系。而 race 的参数跟 all 一致，只不过 race 只会返回一个最快完成的 Promise，也就是谁快就返回谁。

Promise 可以直接通过 new Promise 进行创建，需要传入一个回调，回调有两个参数，resolve 用来返回正确的值，而返回的值将被 then 方法里面的回调捕获；reject 用来返回错误，会被 catch 里面的回调捕获。同时还可以通过 Promise 的静态方法 resolve 和 reject 快速地构建出一个 Promise。

Promise 链一旦开始，返回值会不停地用 Promise 对象重新包裹，只能不停地“then”。比如上面的代码 return 20，只能用 then 的回调获取 20 这个值。

更多关于 Promise 的详细内容可以在 <http://liubin.org/promises-book/> 找到，而对于 Promise 如何实现，可以在 <https://www.promisejs.org/implementing/> 找到。

6. async/await “大杀器”

Node.js 从 8.9 版本之后就支持 async/await 关键字了，假如是在前端，则可以使用 babel 转义 async/await 关键字。异步的本质是回调，而 async/await 可以在语法层面上规避回调，代码如下：

```
async function func() {  
  return 2;  
}  
  
func().then(console.log);
```



```
const getPosts = () =>
  new Promise((resolve, reject) => {
    resolve([
      {
        name: "a"
      },
      {
        name: "b"
      },
      {
        name: "c"
      },
      {
        name: "d"
      }
    ]);
  });

async function func2() {
  try {
    const number = await func();
    const posts = await getPosts();
    console.log(number);
    console.log(posts);
  } catch (e) {
    console.log(e);
  }
}

func2();
```

运行结果如下：

```
2
2
[ { name: 'a' }, { name: 'b' }, { name: 'c' }, { name: 'd' } ]
```

`async` 修饰 `function`，说明这是一个异步的方法，比如 `func` 函数，当调用的时候，它会返回一个 `Promise`，只有通过 `then` 方法才能获取返回的 2。

对于异步 `func2`，我们用到了 `await` 关键字，当调用 `func` 异步函数的时候，返回 `Promise`，而把 `await` 放在 `Promise` 的前面，就可以获取被 `Promise` 包裹的值，比如 `number` 就等于 2。

通常 `async` 与 `await` 成对出现，`await` 后面可以跟 `Promise` 和其他 `async` 函数，也可以跟普通的同步函数。假如其后跟的是普通的同步函数，则行为跟普通同步函数一样。

使用 `async/await` 之后，就可以直接通过 `try/catch` 来捕获错误。

7. 小结

现在我们实现异步编程的选择是 `async/await` 加上 `Promise`，那么此时读者一定会有这样的疑问，使用 `Promise` 如何兼容以前的回调呢？有两种方式解决，一是自己封装，二是通过 `npm` 安装其他人封装好的，代码如下：

```
const fs = require("fs");

const readFilePromise = filename =>
  new Promise((resolve, reject) => {
    fs.readFile(filename, (err, data) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });

async function main() {
  const txt = await readFilePromise("mock.txt");
  console.log(txt.toString());
}

main();
```

2.2 Koa.js 基础知识

Koa.js 是一个非常小巧的 Node.js 服务器，实现了中间件的模式，所以可以对 Koa.js 进行

任意扩展。Koa.js 本身只自带中间件与请求响应的一些帮助方法，甚至连路由、视图渲染、数据库模型都没有，可见 Koa.js 是多么精简。但是这并不意味着 Koa.js 不够强大，Koa.js 拥有非常多的中间件，各种服务以中间件的形式给 Koa.js 增加功能，所以 Koa.js 就像乐高积木一样，可以任意组合各种功能。

想要深入学习 Koa.js 的读者，可以到 <https://nodelover.me/course/deep-into-koa> 免费获取 Koa.js 的源码阅读教程。也可以到 <https://nodelover.me/course/koa-todo> 获取使用 Koa.js 创建一个 RESTful API 服务的实践教程，以及到 <https://nodelover.me/course/middleware> 了解 Express.js 与 Koa.js 中间件的异同。

2.2.1 Koa.js 中间件核心代码

下面的代码来自代码仓库 <https://github.com/koajs/compose>:

```
function compose (middleware) {
  return function (context, next) {
    // last called middleware #
    let index = -1
    return dispatch(0)
    function dispatch (i) {
      if (i <= index) return Promise.reject(new Error('next() called multiple times'))
      index = i
      let fn = middleware[i]
      if (i === middleware.length) fn = next
      if (!fn) return Promise.resolve()
      try {
        return Promise.resolve(fn(context, function next () {
          return dispatch(i + 1)
        })))
      } catch (err) {
        return Promise.reject(err)
      }
    }
  }
}
```

然后我们来测试一下，代码如下：


```
async function a(ctx, next) {  
  console.log(1);  
  const hello = await Promise.resolve("hello node.js");  
  console.log(hello);  
  await next();  
  console.log("a end");  
}  
  
async function b(ctx, next) {  
  console.log(2);  
  const hello = await Promise.resolve("hello node.js");  
  console.log(hello);  
  await next();  
  console.log("b end");  
}  
  
compose([a, b])({});
```

结果如下:

```
1  
hello node.js  
2  
hello node.js  
b end  
a end
```

通过 `async/await` 语法可以非常优雅地进行异步编程, 为什么结果会像上面所示呢? 因为 `next` 函数返回下一个异步函数的 `Promise`, 即异步函数 `b`, 而在异步函数 `a` 中 `await next()` 就会等待异步函数 `b` 执行完, 而 `b` 函数中又执行 `await next()`, 此时 `b` 函数的 `next` 没有下一个中间件, 根据 `compose` 方法的代码, 当不存在 `fn` 的时候, 就是一个空的 `Promise` 即 `Promise.resolve()`。然后执行 `console.log("b end")`。

现在异步函数 `b` 已经执行完毕, 所以在 `a` 函数中就不用再等待了, 直接执行 `console.log("a end")`, 这就是为什么 `"b end"` 会在 `"a end"` 之前。

简单地说, 等价于以下代码, 运行结果跟之前一样。

```
async function a_new(ctx) {
```

```
console.log(1);
const hello = await Promise.resolve("hello node.js");
console.log(hello);
await b_new(ctx);
console.log("a end");
}

async function b_new(ctx) {
  console.log(2);
  const hello = await Promise.resolve("hello node.js");
  console.log(hello);
  console.log("b end");
}

a_new({});
```

这样的中间件连接起来就像一个洋葱圈，如图 2-1 所示。

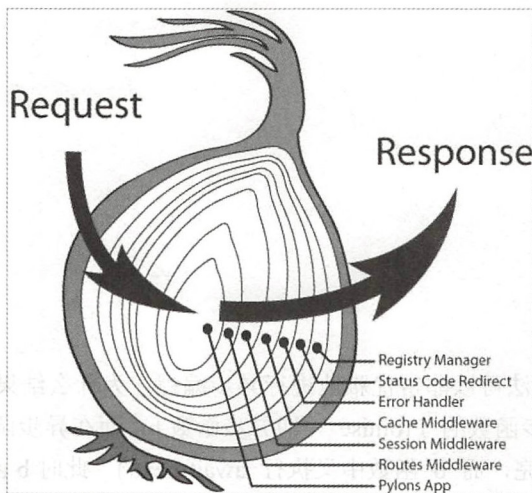


图 2-1

2.2.2 Koa.js 插件

Koa.js 插件可以到 <https://github.com/koajs/koa/wiki> 获取。接下来我们学习如何使用中间件来创建一个自己的中间件。

1. 使用中间件

当我们在 Wiki 页面找到一个想要的功能中间件的时候，一定要查看它的 README.md 文档，因为它会告诉你如何使用该中间件。

- 创建一个文件夹

```
mkdir useragent && cd useragent
```

- 安装插件

```
npm install -S koa koa-useragent
```

- 新建 index.js

```
const Koa = require("koa");

const app = new Koa();

const userAgent = require("koa-useragent");

app.use(userAgent);

app.use(async (ctx, next) => {
  console.log(require("util").inspect(ctx.userAgent));
});

app.listen(3000);
```

- 运行服务器

```
node index.js
```

- 打开浏览器

输入 127.0.0.1:3000，结果如下：

```
{
  isAuthoritative: true,
  isMobile: false,
  isTablet: false,
  ....
}
```



```
browser: 'Chrome',
version: '63.0.3239.132',
os: 'macOS Sierra',
platform: 'Apple Mac',
geoIp: {}
}
```

2. 创建一个中间件

每一个中间件会接收 `ctx` 和 `next` 两个参数，`next` 是下一个回调的 `Promise`，而 `ctx` 是 Koa 封装的上下文，这个对象包含响应和请求的所有方法，当我们想要添加一些方法的时候，可以直接挂载到 `ctx` 对象上。

现在我们来创建一个在控制台上输出当前访问 URL 的中间件。

- 创建 `log.js`

```
module.exports = options => {
  if (!options.format) {
    console.error("需要传递 format 函数");
  }
  return async (ctx, next) => {
    console.log(options.format(ctx.url));
    await next()
  };
};
```

- 修改 `index.js`

```
const Koa = require("koa");
const userAgent = require("koa-useragent");
const log = require("./log");

const app = new Koa();

const config = { format: text => `===== ${text} =====` };
app.use(userAgent);
app.use(log(config));

app.listen(3000);
```



- 启动服务器

```
node index.js
```

- 访问
 - `http://127.0.0.1:3000/some`
 - `http://127.0.0.1:3000`

结果如下：

```
===== /some =====
===== / =====
```

通常自定义的中间件需要传递配置项，所以开发的中间件插件通常是两层函数，一层函数用来传递配置项，一层是 Koa.js 中间件，前面代码中就传递了一个 `format` 格式化函数。注意一定要调用 `next`，否则无法调用后续的中间件。

3. 小结

现在我们对 Koa.js 有了一个大概的了解，不对 Koa.js API 进行阐述，是因为后面使用的 Egg.js 对 Koa.js 进行了封装，Egg.js 提供了更多好用的 API 与更高的扩展性。

2.3 Egg.js 基础知识

Egg.js 由 Koa.js 扩展而来，添加了多进程支持，并且参考了 Ruby On Rails 的设计哲学，以约定优先的配置。

更多的目录约定可以在 <https://eggjs.org/zh-cn/basics/structure.html> 中找到。

- `app/router.js`，路由映射配置；
- `app/controller`，存放控制器的目录，用来处理跳转相关的逻辑；
- `app/service`，用来存放业务逻辑；
- `app/middleware`，存放中间件的目录；
- `app/public`，存放静态文件的目录；
- `app/extends`，扩展框架目录，比如在 `ctx` 上添加一些变量方法等；
- `config`，配置目录、中间件的配置项、环境配置变量等；
- `test`，测试文件目录；



- `app.js`，可以在该文件内添加启动钩子。

2.3.1 实现 egg-core

Egg.js 能保持这些约定，得益于 `egg-core` 中的各种 Loader，比如 `context_loader`、`file_loader`，现在我们来实现一个简易版的 `egg-core`。

- 初始化项目

```
mkdir egg-core && cd egg-core && npm init -y
```

- 安装依赖

```
npm i koa globby
```

`globby` 是搜索目录下所有文件的一个工具，主要用来搜索约定目录下的 JS 文件。

- 创建 `egg.js`

首先构建目录映射变量，即 `fileAbsolutePath`，把 `key` 作为 `ctx` 上挂载的名称，而 `value` 则是目录的路径，把这个路径传递给 `globby`，获取该目录下的所有文件，然后逐个地通过 `require` 导入模块。根据目录的不同进行不同的载入，`config` 直接挂载到 `ctx` 上；`service` 也挂载到 `ctx` 上；`middleware` 通过 `use` 进行载入，并从 `ctx.config` 上读取相应的配置项，要求与 `middleware` 下面的文件名相同。

```
const { resolve, join, parse } = require("path");
const globby = require("globby");

module.exports = app => {
  const AppPath = resolve(__dirname, "app");
  const context = app["context"];
  // 方法一
  const fileAbsolutePath = ["config", "middleware", "service"].reduce(
    (folderMap, v) => ((folderMap[v] = join(AppPath, v)), folderMap),
    {} // 初始值
  );

  // 方法二
  // const fileAbsolutePath = {
  //   config: join(AppPath, "config"),
  //   middleware: join(AppPath, "middleware"),
  //   service: join(AppPath, "service"),
  // };
}
```




```

// 挂载 middleware: join(AppPath, "middleware"),
//    service: join(AppPath, "service")
// };

Object.keys(fileAbsolutePath).forEach(v => {
  const path = fileAbsolutePath[v]; // 对应的路径
  const prop = v; // 挂载到 ctx 上面的 key
  const files = globby.sync("**/*.js", {
    cwd: path
  });
  if (prop !== "middleware") {
    context[prop] = {}; // 初始化对象
  }

  files.forEach(file => {
    const filename = parse(file).name; // 文件的名称作为 key 挂载到子对象上面
    const content = require(join(path, file)); // 导入内容
    // middleware 处理逻辑
    if (prop === "middleware") {
      if (filename in context["config"]) {
        // 先传递配置选项
        const plugin = content(context["config"][filename]);
        app.use(plugin); // 加载中间件
      }
      return;
    }
    // 配置文件处理
    if (prop === "config" && content) {
      context[prop] = Object.assign({}, context[prop], content);
      return;
    }

    context[prop][filename] = content; // 挂载 service
  });
});
};

```

- 新建 index.js



现在对之前写好的精简版 egg-core 进行测试。真正的 egg-core 还支持框架载入，这里为了健壮性，代码复杂了很多，不过其底层原理还是载入文件：

```
const Koa = require("koa");
const init = require("./egg");

const app = new Koa();

init(app);

app.use(async (ctx, next) => {
  console.log(ctx.service);
  console.log(ctx.config);
  ctx.type = "application/json";
  ctx.body = ctx.service.user.getUser();
});

app.listen(3000);
```

- 新建 app/config/index.js

这是配置文件，log 下面是名为 log 的中间件的配置项。

```
module.exports = {
  log: {
    format: txt => `====${txt}====`
  }
};
```

- 新建 app/config/other.js

```
module.exports = {
  app_name: "egg"
};
```

- 新建 app/middleware/log.js

```
module.exports = config => async (ctx, next) => {
  console.log(config.format(ctx.url));
```



```
    await next();  
  };
```

- 新建 app/service/user.js

```
module.exports = {  
  getUser() {  
    return [  
      {  
        name: "A"  
      },  
      {  
        name: "B"  
      },  
      {  
        name: "C"  
      }  
    ];  
  }  
};
```

- 运行服务

```
node index.js
```

- 打开 127.0.0.1:3000

结果如图 2-2 所示，笔者使用百度的 FeHelper 浏览器插件对 JSON 进行了自动格式化。

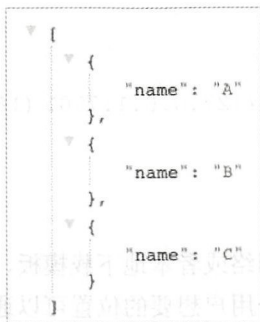


图 2-2

控制台的输出如下：




```
====/====
```

```
{ user: { getUser: [Function: getUser] } }  
{ log: { format: [Function: format] }, app_name: 'egg' }
```

现在我们写的 egg-core 可以正常工作了，已经正确加载了配置、服务和中间件。

2.3.2 实现 egg-init

1. 使用 egg-init

首先创建 egg 新项目，体验一下 egg-init，egg-init 是创建 egg 项目的命令行工具。

- 安装 egg-init

```
npm i egg-init -g
```

- 初始化项目

系统会提示你填写一些信息：

```
egg-init first_app --type=simple
```

- 安装依赖

```
cd first_app && npm install
```

- 运行项目

```
npm run dev
```

当看到 egg started on http://127.0.0.1:7001 (16404ms) 时，打开 127.0.0.1:7001 即可访问 egg.js 的服务。

2. egg-init 的原理

模板类的脚手架的原理就是从网络或者本地下载模板，然后进行模板变量替换，替换完成之后放在用户想要的位置即可。这个用户想要的位置可以通过命令行来指定，所以通常还需要一个命令行解析工具。

官方的 egg-init 复制文件与模板替换功能是通过 mem-fs、mem-fs-editor 来实现内存中的复制和修改的，而命令行解析通过 yargs 模块实现。



官方的 egg-init 源码很多，而且它是通过网络下载模板进行下载的，我们换一个类似且更简单的项目源码来阅读，它叫 module，它通过 vinyl-fs 来进行文件的复制。用过前端打包工具 gulp 的读者一定对 vinyl-fs 非常熟悉，用 gulp 复制文件其实非常简单。解析命令行工具用的是 yargs。

项目地址为 <https://github.com/lukehorvat/module>。

首先我们看一下 index.js:

```
import path from "path";
import concat from "concat-stream";
import template from "lodash.template";
import map from "map-stream";
import fs from "vinyl-fs";
export default createModule;

function createModule(dir) {
  return new Promise((resolve, reject) => {
    fs
      .src(path.resolve(__dirname, "..", "templates", "*", "*"), { dot: true })
      .pipe(renameFiles({ gitignore: ".gitignore" })) // See:
      .pipe(templateFiles({ name: path.basename(dir) }))
      .once("error", reject)
      .pipe(fs.dest(dir))
      .pipe(collectFiles(resolve));
  });
}

function renameFiles(renames) {
  return map((file, cb) => {
    if (file.basename in renames) {
      file.basename = renames[file.basename];
    }
    cb(null, file);
  });
}

function templateFiles(data) {
```



```
return map((file, cb) => {
  file.contents = new Buffer(template(file.contents)(data));
  cb(null, file);
});
```

然后看一下导入了哪些模块：

- `concat-stream`，用来获取流中的内容，把所有的内容连接起来一次返回；
- `lodash.template` `lodash`，字符串替换函数，比如 `<%= name %>` 会被替换为 `name` 变量的值；
- `map-stream`，对流中每一次发出的值进行处理；
- `vinyl-fs`，把文件转化为流处理。

接下来看实现过程：

`createModule` 接收一个 `dir` 路径，返回一个 `Promise`，说明它是一个异步函数。传递给 `fs.src` 一个绝对路径与配置，`fs.src` 内部其实就是使用 `glob` 模块匹配该绝对路径下的文件，这里用的是 `**/*`，说明匹配上级目录 `template` 下的所有文件。

然后通过 `renameFiles` 函数进行文件重命名，因为以点开头的文件在 `Linux` 下面是隐藏文件，所以作为模板不会有以点开头的文件，而实际环境中需要以点开头，所以要重命名。`glob` 匹配返回的是一个路径数组，我们通过 `map` 获取其中每一项，看哪一项是需要重命名的。

`renameFiles` 与 `templateFiles` 都是高阶函数，因为要传递配置项，所以都是高阶函数。与之前的中间件类似，之前 `Koa.js` 接收的参数为 `ctx`、`next`，而这里通过 `map` 处理后的回调参数为 `file`、`cb`，这里采用 `callback` 回调的形式，当调用这个 `cb` 时说明这个异步完成了，第一个参数是错误，第二个参数是继续传递给后面流的内容，这里的 `file` 经过 `fs` 处理后，它会有 `basename`、`contents` 等属性，分别代表文件名与文件内容，然后对这两个属性进行处理就行了。例如，重命名与模板替换。

最后通过 `fs.dest(dir)` 将内容输出到 `dir` 目录，即将内容复制过去。`collectFiles` 会将流中匹配到的文件数组取出，通过 `resolve` 传递出去。`collectFiles` 其实可有可无，因为在 `fs.dest` 中就已经复制完成了。

下面看一下 `cli.js`：

```
#!/usr/bin/env node

import module from "./";
import pkg from "../package.json";
```




```
import path from "path";
import chalk from "chalk";
import tildify from "tildify";
import yargs from "yargs";

const {argv} =
  yargs
    .usage(`Usage: ${chalk.cyan(pkg.name, chalk.underline("<dir>"))}`)
    .demand(0, 1, chalk.red("Too many directories specified. "))
    .option("h", { alias: "help", describe: "Show help", type: "boolean" })
    .option("v", { alias: "version", describe: "Show version", type: "boolean" });

if (argv.help || argv.h) {
  yargs.showHelp();
  process.exit();
}

if (argv.version || argv.v) {
  console.log(pkg.version);
  process.exit();
}

Promise.resolve(
  path.resolve(process.cwd(), argv._.length > 0 ? String(argv._[0]) : ".")
).then(dir => {
  console.log(chalk.green("Creating module..."));
  return module(dir);
}).then(files => {
  files.map(tildify).forEach(file => console.log(chalk.green("+", file)));
  console.log(chalk.green("Module created!"));
  process.exit();
}).catch(() => {
  console.error(chalk.red("An error occurred. "));
  process.exit(1);
});
```

`#!/usr/bin/env node` 这行命令其实是告诉系统，用 `node` 来执行该文件，假如给该文件加上执行权限，则可以直接运行该文件了。



看看用了哪些模块：

- `tildify`，有的时候文件绝对路径太长了，假如目录在用户目录下面，那么就可以将前面的路径替换为~；
- `yargs`，解析命令行的一些参数；
- `chalk`，给输出内容加上色彩，其实输出的色彩就是加上一些颜色状态码，可以用 `console.log` 输出 `\u001B[94m Blue` 这段代码试一试。

首先通过 `yargs` 的 `usage` 指定如何使用该命令行的帮助信息，`demand` 则是指定接收的参数长度，因为只能接收一个目录地址，所以长度不能超过 1，通过 `option` 指定选项，比如 `v` 就是 `version`，意为打印出版本，而 `h` 则是打印出帮助信息。然后得到 `argv` 变量。

`argv.help` 和 `argv.h` 可以判断用户是否输入了 `module -help` 或者 `module -h`，而 `argv._` 则是一个数组，比如用户输入 `module a b c`，那么 `argv._` 就是 `['a', 'b', 'c']`，因为这里指定了 `demand`，最多只能输入一个参数，输出多个会报错。

通过 `process.cwd()` 获取当前运行该命令的目录，看用户是否输入了目录，从 `argv._[0]` 中获取，假如没有就默认是当前目录。然后调用 `module` 复制文件，将文件彩色输出即可。

2.3.3 实现 egg-cluster

这一节我们实现一个多进程的 HTTP 服务，并且添加热重启功能。官方实现是通过 `egg-development` 插件监听文件变化，当文件变化的时候，触发 `egg-cluster` 的 `reload` 事件进行重启。

首先将之前的 `egg-core` 源码复制一份，名字修改为 `egg-cluster`。

- 安装依赖

```
npm i cfork chokidar cluster-reload
```

- 创建 `cluster.js`

```
const cfork = require("cfork");
var chokidar = require("chokidar");
const { resolve } = require("path");
const reload = require("cluster-reload");
```

```
const master = cfork({
  exec: resolve(__dirname, "index.js"),
```

```

    count: 2
  });

chokidar.watch("./app").on("change", (event, path) => {
  console.log(event, path);
  reload(2);
});

```

通过 `cfork` 我们可以多进程地运行 `index.js`，然后利用 `chokidar` 监听代码文件，当发生修改时，通过 `cluster-reload` 重启进程。

- 运行服务器

```
node cluster.js
```

- 查看 127.0.0.1:3000

会看到如图 2-3 所示的内容。

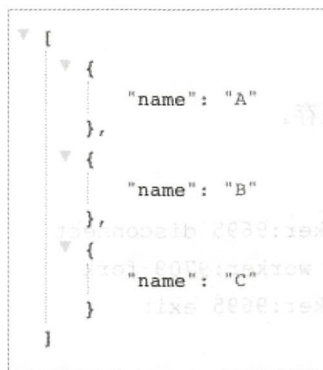


图 2-3

- 查看进程

```
ps aux | grep node
```

可以看到除了 `node cluster.js` 进程，还有两个运行 `index.js` 的进程：

```

Yugo      10306  0.0  0.4  3063776  33100 s000  S+   2:41下午   0:00.20 /usr/local/bin/
node /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter4/4-3/egg-cluster/index.js
Yugo      10305  0.0  0.4  3063776  33384 s000  S+   2:41下午   0:00.21 /usr/local/bin/
node /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter4/4-3/egg-cluster/index.js
Yugo      10304  0.0  0.5  3115504  44528 s000  S+   2:41下午   0:00.40 node cluster.js

```

- 修改 service/user.js

```
module.exports = {
  getUser() {
    return [
      {
        name: "A"
      },
      {
        name: "B"
      },
      {
        name: "C"
      },
      {
        name: "D"
      }
    ];
  }
};
```

给 service 添加一些内容并保存。

- 重启信息

```
[cfork:master:9692] worker:9695 disconnect
[cfork:master:9692] new worker:9709 fork
[cfork:master:9692] worker:9695 exit
```

如上所示的重启信息意思是，进程号为 9695 的进程断开连接，创建了新的进程 9709，进程 9695 退出。这样我们就实现了重启。

- 再次刷新查看 localhost:3000

如图 2-4 所示，已经添加了 D。



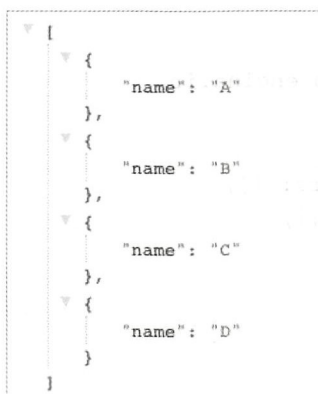


图 2-4

2.4 Egg.js 插件

假如想要获得一些关于 Egg 实例的视频可以到 <https://nodelover.me/course/egg-example> 免费获取。

2.4.1 egg-socket.io

源码地址为 <https://github.com/eggjs/egg-socket.io>，我们通过 egg-socket.io 这个例子来学习 egg 插件，首先要学会如何使用这个插件，然后运行一个实例加以实践。

官方的 socket.io 插件文档在 <https://eggjs.org/zh-cn/tutorials/socketio.html> 可以找到。socket.io 是 Websocket 协议在 Node.js 的实现，即长连接，常见的应用场景就是在线聊天。

1. 如何使用

通过它的源码地址可以看到英文简介，其实它有中文简介，打开 Readmd.zh_CN.md 即可。假如我们要使用一个 Egg 插件，则第一步是通过 NPM 进行安装，然后在 config/plugin.js 下告诉 Egg 开启哪些插件。通常插件都会有一些配置项，在 config/config.default.js 中配置即可。

启动插件的配置如下（文件在 config/plugins.js 中）：

```
exports.io = {
  enable: true,
  package: 'egg-socket.io',
};
```

egg-socket.io 的默认配置如下（文件在 config/config.default.js 中）：


```
exports.io = {
  init: { } // passed to engine.io
  namespace: {
    '/': {
      connectionMiddleware: [],
      packetMiddleware: [],
    },
  },
  redis: {
    host: '127.0.0.1',
    port: 6379
  }
};
```

`init` 中是初始化的选项，而 `namespace` 则是命名空间，可以把命名空间理解为路由的 URL，每一个命名空间下有两个中间数组配置项，分别是连接时的中间件和通信时的中间件。假如你的服务是集群的，由多台服务器同时提供服务，那么可能需要 Redis 来实现同步数据。

2. 运行实例

其实官方的源码提供了实例。

- 复制源码

```
git clone https://github.com/eggjs/egg-socket.io.git
```

- 安装 `egg-socket.io` 依赖

```
cd egg-socket.io
```

```
npm i
```

- 安装 `example` 的依赖

```
cd example
```

```
npm i
```

```
npm i ..
```

- 查看 `example/package.json`

```
"egg-socket.io": "file:..",
```

先安装其他依赖，然后把 `egg-socket.io` 替换为本地依赖，这样会把 `egg-socket.io` 软连接到该项目的 `node_modules` 下。`file:` 的意思就是本地，而上级 `..` 目录就是 `egg-socket.io`，可以在修改代码后重新安装，便于调试学习。

- 在 `example` 下运行服务端

```
npm run start
```

- 在 `example` 下启动客户端

```
node clinet.js
```

- 服务器输出

```
[ 'chat', 'hello world!' ]
chat : hello world! : 4701
packet response!
```

- 客户端输出

```
connect!
res from server: auth!Helle Man!!
res from server: packet!Helle Man!!
res from server: Helle Man!!
```

- `client.js` 代码

```
const socket = require('socket.io-client')('http://127.0.0.1:7001');

socket.on('connect', () => {
  console.log('connect!');
  socket.emit('chat', 'hello world!');
});

socket.on('res', msg => {
  console.log('res from server: %s!', msg);
});
```

首先通过 `socket.io-client` 与 `http://127.0.0.1:7001` 建立 WebSocket 连接。

- 后端的 socket.io 服务配置

```
exports.io = {
  namespace: {
    '/': {
      connectionMiddleware: [ 'auth' ],
      packetMiddleware: [ 'filter' ],
    },
    '/chat': {
      connectionMiddleware: [ 'auth' ],
      packetMiddleware: [],
    },
  },
};
```

上面代码的意思是经过 / 的时候要通过 auth 中间件，发送消息的时候要通过 filter 中间件，而经过 /chat 的时候也要通过 auth 中间件。

在 client.js 中得到 Socket 变量的时候就建立了连接，对于服务器而言，当发生连接的时候进入 auth 中间件，在这个中间件里会向客户端发送 auth!Helle Man!!，在客户端会被 socket.on('res') 事件接收并打印出来。auth 中间件代码如下：

```
module.exports = () => {
  return async (ctx, next) => {
    const say = await ctx.service.user.say();
    ctx.socket.emit('res', 'auth!' + say);
    await next();
    console.log('disconnect!');
  };
};
```

建立连接会触发 socket.on('connect') 中的回调，打印出 connect!，然后向 / 的 chat 中发送消息 hello world!。

- 查看路由

```
module.exports = app => {
  // app.io.of('/')
  app.io.route('chat', app.io.controller.chat.index);
};
```

```
// app.io.of('/chat')
app.io.of('/chat').route('chat', app.io.controller.chat.index);
};
```

of 说明在 / 路径下, socket.emit('chat', 'hello world!') 的第一个参数 chat 代表路由的 route('chat'), 当执行 emit('chat', xxxx) 的时候, 会触发 app/io/ctrlonller/chat.js 中的 index 方法, 但是在触发之前还要看一下有没有配置中间件, 有中间件则先进入中间件。

- 查看 filter 中间件

从之前服务端的配置知道经过 / 的消息会经过 filter 中间件, 而 filter 中间件如下:

```
module.exports = () => {
  return async (ctx, next) => {
    console.log(ctx.packet);
    const say = await ctx.service.user.say();
    ctx.socket.emit('res', 'packet!' + say);
    await next();
    console.log('packet response!');
  };
};
```

首先会打印 ctx.packet 输出 ['chat', 'hello world!'], 然后向客户端发送 packet!Helle Man!!, 客户端会输出 res from server: packet!Helle Man!!, 然后 next 会进入控制器, 即 chat 控制器的 index 方法。

- 查看 chat.js 控制器

```
module.exports = app => {
  class Controller extends app.Controller {
    async index() {
      const message = this.ctx.args[0];
      console.log('chat :', message + ' : ' + process.pid);
      const say = await this.ctx.service.user.say();
      this.ctx.socket.emit('res', say);
    }
  }
  return Controller;
};
```


控制器其实有两种写法，可以导出一个函数，也可以导出一个类，官方目前推荐第二种写法，第一种写法是旧的写法，不过同样支持。

- 第一种

```
module.exports = app => {
  class PostController extends app.Controller {
    // ...
  }
  return PostController;
};
```

- 第二种

```
const Controller = require('egg').Controller;
class PostController extends Controller {
  // ...
}
module.exports = PostController;
```

控制器首先从 `this.ctx.args[0]` 中获取 `hello world`，输出 `chat : hello world! : 4701`，4701 是进程号，然后向客户端发送 `Helle Man!!`。

执行完回到 `filter` 中间件的 `next()` 位置，然后输出 `packet response!`。

- 结束客户端

使用 `Ctrl+C` 快捷键结束客户端，可以看到服务端输出 `disconnect!`，说明连接断开了。

- 修改 `client.js` 的连接地址

使用 `/chat` 命名空间：

```
const socket = require('socket.io-client')('http://127.0.0.1:7001/chat');
```

- 查看客户端输出

```
connect!
res from server: auth!Helle Man!!
res from server: Helle Man!!
```

- 查看服务端输出



```
chat : hello world! : 4701
```

由于 /chat 命名空间的配置不通过 filter 中间件，所以就少了服务端的两句输出和发送到客户端的 packet!Helle Man!!。

```
[ 'chat', 'hello world!' ]
packet response!
```

2.4.2 原理解读

下面介绍实现 egg-socket.io 的原理，但是并非逐行阅读源代码，而是将核心实现原理讲解清楚，对于中间过程穿插的 socket.io 知识，笔者也会做一些讲解。

1. 目录约定

在 egg-socket.io 中有中间件的概念，分为连接阶段的中间件与发包的中间件。这个中间件的代码从何处加载？这就是第一个要实现的逻辑。

在 Egg.js 中，可以加载插件、框架与应用，每一个插件、框架与应用都是一个加载单元，我们实现了在加载单元中的目录约定，即在 app/io/middleware 下存放中间件，在 app/io/controller 下存放控制器，那么每一个加载单元都可能有些文件，所以要获取所有的加载单元，然后跟约定好的目录拼接成绝对路径，加载其中的文件。

```
let dirs = app.loader.getLoadUnits().map(unit => path.join(unit.path, 'app',
'io', 'middleware'));
```

通过 app.loader.getLoadUnits() 可以获取所有的加载单元，将 unit.path 与约定的目录拼接就行了。

```
app.io.middleware = app.io.middleware || {};
new app.loader.FileLoader({
  directory: dirs,
  target: app.io.middleware,
  inject: app,
}).load();
```

然后初始化 app.io.middleware，通过 FileLoader 将目录下的文件加载到 target 对象中。

对于 controller 也是同样的原理，只不过调用的是 app.loader.loadController 方法，因为加



载控制器官方已经定义好了，按照官方处理控制器的逻辑加载到 `app.io.controller` 中。

```
app.io.controller = app.io.controller || {};  
app.loader.loadController({  
  directory: dirs,  
  target: app.io.controller,  
});
```

这些中间件是在 `config.default.js` 中配置的，所以我们需要做的就是从 `app.config.io` 取得配置项，而 `app.config` 变量中有 `config.default.js` 所有的配置项，`egg-socket.io` 默认的配置项是 `io`。

遍历 `namespace` 后经过一些处理，可以从 `app.io.middleware` 中获取中间件的执行逻辑，该执行逻辑是一个数组，通过 `koa-compose` 把这个数组中的中间件串联起来，就可以进行初始化了。

2. 添加建立连接中间件

```
nsp.use((socket, next) => {  
  connectionMiddlewareInit(app, socket, next, connectionMiddlewares);  
});
```

`nsp` 就是 `socket.of()` 返回的变量，即通过 `socket.io` 的 `of` 方法可以指定命名空间，然后通过 `use` 添加中间件。因为此时还没有建立连接，所以这个是建立连接的中间件。

3. 添加发包中间件

```
nsp.on('connection', socket => {  
  socket.use((packet, next) => {  
    packetMiddlewareInit(app, socket, packet, next, packetMiddlewares, nsp);  
  });  
})
```

可以看到这个是放到 `connection` 事件中的发包的中间件，这里调用了 `xxxInit` 方法。

4. 中间件初始化

调用这些中间件时，有一点要注意的是，`socket.use` 有一个 `next` 参数，即 `socket.io` 自带中间件，我们通过 `koa-compose` 组合也有一个 `next`，所以要对这两个 `next` 进行组合。组合方法如下：



```
composed(ctx, async () => {
  next();
  nexted = true;
  // after socket emit disconnect, resume middlewares
  await new Promise(resolve => {
    socket.once('disconnect', reason => {
      debug('socket disconnect by: %s', reason);
      resolve(reason);
    });
  });
})
.then(() => !nexted && next())
.catch(e => {
  next(e); // throw to the native socket.io
  app.coreLogger.error(e);
});
```

`composed` 就是连接好的中间件，第二个参数的回调是指这个中间件串调用完，会把这个回调作为最后一个 `next`，而这个 `next` 刚好接上 `socket.io` 的 `next`。这里没有 `await socket.io` 的 `next`，是因为它与 `socket.io` 的 `next` 是同步的。

- `socket.io` 中间件的实现逻辑

```
Socket.prototype.run = function(event, fn){
  var fns = this.fns.slice(0);
  if (!fns.length) return fn(null);

  function run(i){
    fns[i](event, function(err){
      // upon error, short-circuit
      if (err) return fn(err);

      // if no middleware left, summon callback
      if (!fns[i + 1]) return fn(null);

      // go on to next
      run(i + 1);
    });
  }
```




```
}
```

```
run(0);
```

```
};
```

- koa 中间件的实现逻辑

```
function (context, next) {  
  // last called middleware #  
  let index = -1  
  return dispatch(0)  
  function dispatch (i) {  
    if (i <= index) return Promise.reject(new Error('next() called multiple  
times'))  
    index = i  
    let fn = middleware[i]  
    if (i === middleware.length) fn = next  
    if (!fn) return Promise.resolve()  
    try {  
      return Promise.resolve(fn(context, function next () {  
        return dispatch(i + 1)  
      })))  
    } catch (err) {  
      return Promise.reject(err)  
    }  
  }  
}
```

从上面代码可以看出，Koa 的实现逻辑用了 Promise 做包裹，所以是异步的。而这里 await 了一个 Promise，这是一个只有在连接 disconnect 时才会调用的 Promise。也就是说，只有将这个连接 disconnect 了，才会执行中间件中写在 next() 之后的逻辑。

例子中的 auth.js 代码如下：

```
module.exports = () => {  
  return async (ctx, next) => {  
    const say = await ctx.service.user.say();  
    ctx.socket.emit('res', 'auth!' + say);  
    await next();  
  }  
}
```



```
    console.log('disconnect!');  
  };  
};
```

只有 socket disconnect 事件发送之后，才会打印出 disconnect。而对于发包的中间件也有类似的 await，不过是 finished 事件而已。

5. 命名空间与事件名称（路由）

我们在 router.js 中指定的事件名称与控制器的映射都在 RouterConfigSymbol 中，把这些遍历出来，通过 socket.on 添加上去，这个 socket 是由 nsp 提供的，即命名空间下的 socket，当发生 event 事件时，调用控制器的 handler。

```
for (const [event, handler] of nsp[RouterConfigSymbol].entries()) {  
  socket.on(event, (...args) => {  
    handler.call(ctx).catch(e => {  
      .....  
    })  
  })  
}
```

6. 添加帮助方法

首先添加 app.io 变量，实例化 socket.io 的实例，然后给 io 添加 route 方法，这个方法会把上面代码的 event 和 handler 添加到 nsp[RouterConfigSymbol] 中。

2.5 制作一个 Egg.js 插件

现在我们想要制作一个追踪用户行为的插件，把这些行为存储到数据库中，然后把这些数据显示出来，有点类似于百度分析。

1. 初始化项目

```
egg-init --type=plugin egg-tracer  
cd egg-tracer  
npm install
```

通过 egg-init 初始化一个插件模板，然后通过 NPM 安装依赖。

给 package.json 添加一些配置，比如声明依赖：



```
"eggPlugin": {  
  "name": "tracer",  
  "dependencies": ["session"]  
},
```

因为插件要从 session 中读取当前登录的用户信息，所以我们可能依赖 session 中间件，而存储追踪行为数据需要一个存储的地方，所以还可能依赖于 sequelize。sequelize 是 Node.js 非常出名的一个 ORM 模型关系映射工具，即把数据库中的数据映射成一个个 JS 对象。而 Egg.js 开发了对应的 Egg 插件。

由于我们留有接口，可以让使用者自定义存储的地方，所以这里就没有声明 sequelize 了。

2. 声明默认配置

在 config.default.js 中添加一些默认配置，代码如下：

```
exports.tracer = {  
  getUser(ctx) {  
    return ctx.session.user || '';  
  },  
  
  getIp(ctx) {  
    if (ctx.app.config.proxy && ctx.request.ips) {  
      return ctx.request.ips;  
    }  
    return ctx.request.ip || '';  
  },  
  
  async save(ctx, data) {  
    if (ctx.model && ctx.model.Tracer) {  
      return await ctx.model.Tracer.create(data);  
    }  
    return;  
  },  
  
  async auth(ctx) {  
    return true;  
  },  
  
  async getData(ctx) {  
    if (ctx.model && ctx.model.Tracer) {  
      return await ctx.model.Tracer.findAll();  
    }  
  }  
};
```



```
    }  
    return [];  
  },  
  pathUrl: '/tracer/_report'  
};
```

上面定义了一些默认配置，只要使用者不在项目里重新声明，那么都将使用这里的配置。

- `getUser` 是如何拿到用户的回调；
- `getIp` 是如何拿到 IP 的回调；
- `save` 是如何存储数据的回调；
- `auth` 是判断是否可以访问报告页面的回调；
- `getData` 是如何拿到数据的回调；
- `pathUrl` 是报告页面所在的 URL。

3. 创建中间件

新建 `app/middleware/tracer.js`:

```
const useragent = require('useragent');  
const { resolve } = require('path');  
const { readFileSync } = require('fs');  
  
const viewPath = resolve(__dirname, '../view/report.html');  
  
module.exports = ({ getUser, getIp, save, auth, pathUrl, getData }, app) => {  
  const template = readFileSync(viewPath).toString();  
  const report = async ctx => {  
    if (ctx.path === pathUrl) {  
      const can = await auth(ctx);  
      if (can) {  
        ctx.type = 'text/html';  
        const data = await getData(ctx);  
        ctx.body = eval(template);  
        return true;  
      } else {  
        ctx.status = 403;  
        return false;  
      }  
    }  
  }  
};
```



```
    }  
  }  
};  
  
return async (ctx, next) => {  
  const skip = await report(ctx);  
  const agent = (ctx.agent = useragent.parse(  
    ctx.request.header['user-agent']  
  ));  
  await next();  
  if (skip) {  
    return;  
  }  
  const user = await getUser(ctx);  
  const path = ctx.request.path;  
  const ip = await getIp(ctx);  
  const referrer = ctx.request.header['referrer'] || '';  
  const data = {  
    username: user.username || '',  
    user_id: user.id || '',  
    path,  
    referrer,  
    ip,  
    os: agent.os.family,  
    browser: agent.family,  
    device: agent.device.family  
  };  
  try {  
    await save(ctx, data);  
  } catch (e) {  
    ctx.logger.error(e);  
    console.error(e);  
  }  
};  
};
```

这个中间件导出一个函数，第一个对象就是 `config.default.js` 中的配置，所以我们从中解析到一些预先定义好的默认方法。第二个对象是应用 `App` 的实例。

调用这个函数才是返回真正的中间件，中间件的标准格式我们应该也清楚，我们要在中间件里获取数据，需要通过一个叫作 `useragent` 的库进行解析。从 HTTP 头信息上的 `useragent` 字段获取浏览器版本、操作系统版本等。所以需要安装依赖。

```
npm i useragent
```

安装完成之后调用这个库的 `parse` 方法，将 `useragent` 字段传递进去，它就会自动帮我们进行解析，得到一个 `agent` 对象，然后从 `agent.os.family` 中获取操作系统信息，从 `agent.family` 中获取浏览器信息，从 `agent.device.family` 中获取设备信息。

之后通过默认 IP、User 等回调获取到相应的数据，最后通过 `save` 回调将收集到的数据保存起来。

接下来就是显示报告页面，因为在制作插件的时候不能定义路由，所以我们只能通过定义好的 `pathURL` 和 `ctx.path` 进行比对，假如相等就显示出来，当然还要加上一个权限回调，因为不是谁都可以有权限访问的。

`report` 显示报告的函数会返回一个 `skip` 变量，即当前页面是报告页面的时候，不记录到数据中去。

当有权限访问的时候调用 `getData` 获取数据，渲染到模板上，这里用了一个小技巧，即 ES6 的模板字符串，代码如下：

```
const template = `${name}`
const name = "Yugo"
const compiled = eval(template)
```

`compiled` 的内容就是 Yugo。

我们把模板代码放在 `app/view/report.html` 中。

4. 建立视图

新建 `app/view/report.html`，内容如下，一定要注意添加开头和结尾的模板字符串引号。通过 `const dataRaw = ${JSON.stringify(data)}` 这段代码赋值给 `dataRaw` 变量，然后通过 `Vue` 将 `dataRaw` 渲染到 HTML 页面中。这里笔者只做一个示例，更多的统计图表根据个人需求开发即可。

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
```

```
<meta charset="UTF-8">
<title>Tracer</title>
<script>
  const dataRaw = ${JSON.stringify(data)}
  console.log(dataRaw)
</script>
</head>
<body>
  <div id="app">
    <h1>用户追踪</h1>
    <ul>
      <li v-for="row in logs">
        <span>操作系统: {{ row.os }}</span>
        <span>浏览器: {{ row.browser }}</span>
        <span>设备: {{ row.device }}</span>
        <span>用户名: {{ row.username || "匿名用户" }}</span>
        <span>IP: {{ row.ip }}</span>
        <span>路径: {{ row.path }}</span>
      </li>
    </ul>
  </div>

  <script src="https://cdn.bootcss.com/vue/2.5.13/vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        logs: dataRaw
      }
    })
  </script>
</body>
</html>
```

5. 插入中间件

新建 app.js，内容如下：

```
module.exports = app => {
```

```
const index = app.config.coreMiddleware.indexOf('session');
app.config.coreMiddleware.splice(index, 0, 'tracer');
};
```

Egg 内置的中间件在 `coreMiddleware` 中，我们把创建好的 `tracer` 中间件放在 `session` 的后面。

6. 手动测试

在 `test/fixtures/apps/tracer-test` 下有一个自带的测试。

- 修改 `package.json` 文件

```
{
  "name": "tracer-test",
  "version": "0.0.1",
  "dependencies": {
    "egg": "^2.3.0",
    "egg-bin": "^4.3.7",
    "egg-tracer": "file:../../.."
  },
  "scripts": {
    "start": "egg-bin dev"
  }
}
```

- 安装以下依赖

```
npm install
```

通过 NPM 安装依赖。

- 启动插件

在 `config/plugin.js` 中启动插件：

```
module.exports = {
  tracer: {
    enable: true,
    package: 'egg-tracer'
  }
};
```


- 增加配置

在 `config/config.default.js` 中添加配置：

```
exports.keys = '123456';

exports.data = [];

exports.tracer = {
  getUser(ctx) {
    return ctx.session.user || '';
  },

  getIp(ctx) {
    if (ctx.app.config.proxy && ctx.request.ips) {
      return ctx.request.ips;
    }
    return ctx.request.ip || '';
  },

  async save(ctx, data) {
    return ctx.app.config.data.push(data);
  },

  async auth(ctx) {
    return true;
  },

  async getData(ctx) {
    return ctx.app.config.data;
  },

  pathUrl: '/tracer/_report'
};
```

这里直接通过一个数组来模拟存储。

- 运行

```
npm run start
```

运行成功之后，访问 `localhost:7001/`，再访问 `http://localhost:7001/tracer/_report`，会看到如图 2-5 所示的页面。





图 2-5

3 chapter

第 3 章 构建后端 API 服务

3.1 安装相关组件

现在创建 API 后端服务，第一步就是构建 ORM。

1. 初始化

- 安装依赖

```
egg-init --type=simple miao
cd miao
npm install egg-sequelize mysql2 sequelize-cli
```

- 配置 sequelize

下面我们添加一些脚本，构建数据库 ORM：

```
{
  "scripts": {
    "m:new": "sequelize migration:create",
    "m:up": "sequelize db:migrate",
    "m:down": "sequelize db:migrate:undo"
  }
}
```



```

    }
  }
}

```

- 配置 plugin.js 启动插件

```

exports.sequelize = {
  enable: true,
  package: 'egg-sequelize'
};

```

- config.default.js

配置如何连接数据库，这里数据库使用的是关系数据库 MySQL，因为我们要存储的数据基本都是有关系的，使用非关系数据库则不太合适。

```

config.sequelize = {
  dialect: 'mysql',
  database: 'miao',
  host: 'localhost',
  port: '3306',
  username: 'root',
  password: '',
};

```

2. 准备数据库配置

由于 egg-sequelize 的 sequelize-cli 版本有问题，所以我们使用自定义的 sequelize-cli，当然读者也可以 fork 他们的仓库做自己的定制，只需要在 plugin.js 中修改对应的名字就行，这里我们直接 sequelize-cli 就不封装了。

首先我们要做一些配置。

- 创建 .sequelizerc

```

const path = require('path');

module.exports = {
  'config': path.resolve('config', 'config.json'),
  'models-path': path.resolve('app', 'model'),
  'seeders-path': path.resolve('app', 'seeder'),
  'migrations-path': path.resolve('app', 'migration')
};

```



- 创建 config/config.json

定义连接数据库的配置，代码如下：

```
{
  "development": {
    "username": "root",
    "password": null,
    "database": "miao",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "test": {
    "username": "root",
    "password": null,
    "database": "database_test",
    "host": "127.0.0.1",
    "dialect": "mysql"
  },
  "production": {
    "username": "root",
    "password": null,
    "database": "database_production",
    "host": "127.0.0.1",
    "dialect": "mysql"
  }
}
```

- 创建文件夹
 - app/migration
 - app/model
 - app/seeder

migration 是用来存放数据库迁移的文件夹，当我们需要修改表结构的时候，可以通过这些代码来控制数据库表结构的版本回退。

seeder 是用来存放如何生成假数据的文件夹。

model 是表中字段如何映射到 JS 对象的逻辑。



- 创建 user 的 migration

```
./node_modules/.bin/sequelize model:generate --name User
```

它会创建两个文件，接下来我们需要完善其中的逻辑。

```
New model was created at .../miao/app/model/user.js.
```

```
New migration was created at .../miao/app/migration/20180205101006-User.js .
```

这段命令太长了，我们可以使用 Linux 的 alias 命令将其简化一下，假如使用 npm scripts 嵌套太多，则传递参数需要多一层 --，以便把参数解开。

```
alias s="./node_modules/.bin/sequelize model:generate --attributes --name "
```

```
s Post // 即可生成 Post 模型相关文件
```

关于 -- 的问题，可以查看项目下 package.json 的例子：

```
"test": "npm run lint -- --fix && npm run test-local",
```

```
"lint": "eslint ."
```

在 test 中运行 lint，假如想向 eslint 传递 fix 参数，则需要加一个 --，所以使用 alias，不过这个 alias 是当前终端一次性的命令，假如想要永久生效，则需要把这个命令写入 ~/.bashrc 文件中。

读者也可以测试一下之前写的 m:new 命令。

```
npm run m:new --name some
```

```
sequelize migration:create "some"
```

运行的时候发现结果变成了第二条命令。

```
npm run m:new -- --name some
```

如上使用就正常了。

3. Sequelize API

笔者写过一些关于 Sequelize 的电子书，在 <https://www.gitbook.com/@nodelover> 上可以找



到录制过的一些视频，也可以在 <https://nodelover.me/course/sequelize> 上找到，所以笔者将不再对 Sequelize 建模方法进行阐述，大家可以到这个网址找答案，能力强的朋友也可以自己阅读官方文档，总之使用你认为合适的方式进行学习。

4. User 表

至此，我们已经把 User 的相关文件创建好了，现在我们来写逻辑。

创建命令如下，查看有没有建立 alias 别名。

```
s User
// or
./node_modules/.bin/sequelize model:generate --attributes --name User
```

• Migration

```
'use strict';
module.exports = {
  up: (queryInterface, {
    INTEGER, STRING, DATE, TINYINT
  }) => {
    return queryInterface.createTable('Users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: INTEGER
      },
      email: STRING(40),
      password: STRING,
      username: STRING(40),
      weibo: STRING(40),
      weixin: STRING(40),
      team_id: INTEGER,
      receive_remote: TINYINT(1),
      email_verified: TINYINT(1),
      avatar: STRING(40),
      created_at: {
        allowNull: false,
```



```
    type: DATE
  },
  updated_at: {
    allowNull: false,
    type: DATE
  }
});
},
down: (queryInterface, Sequelize) => {
  return queryInterface.dropTable('Users');
}
};
```

以上面的为例，其他模型可参考 <http://github.com/MiYogurt/nodejs-shizhan> 的源码自行创建。这里我们不做过多的约束，假如希望严谨一点，则可自行增加约束。

5. 代码写错了怎么办

有的时候，可能因为一些标点符号导致语法错误，默认是不给出详细的错误调用栈的，想要获得详细错误，应该加上 `debug` 选项，代码如下：

```
./node_modules/.bin/sequelize db:migrate -debug
```

6. 同步到数据库

```
./node_modules/.bin/sequelize db:migrate
```

当同步完成之后，可以在数据库相关的图形化界面里查看表结构，笔者使用的是 `Sequel Pro` 软件，可以发现多了一个 `SequelizeMeta` 表，这是 `Sequelize` 用来记录已经执行了哪些文件的日志。

如何回退版本呢？读者可以通过 `-help` 命令查看 `undo` 相关命令。

7. 添加一些用户数据

```
./node_modules/.bin/sequelize seed:generate --name creat_user
```

这条命令会在 `seeder` 目录下新建一个 JS 文件，然后我们填充代码如下：

```
'use strict';
```




```
const utils = require('utility');
```

```
module.exports = {
```

```
  up: (queryInterface, Sequelize) => {
```

```
    /*
```

```
    Add altering commands here.
```

```
    Return a promise to correctly handle asynchronicity.
```

```
    Example:
```

```
    return queryInterface.bulkInsert('Person', [{
```

```
      name: 'John Doe',
```

```
      isBetaMember: false
```

```
    ]), {});
```

```
  */
```

```
  return queryInterface.bulkInsert('Users', [{
```

```
    email: 'belovedyogurt@gmail.com',
```

```
    password: utils.md5('000000'),
```

```
    inviter_id: 0,
```

```
    username: 'Yugo',
```

```
    weixin: 'xxxx',
```

```
    weibo: 'xxxx',
```

```
    receive_remote: 0,
```

```
    email_verified: 1,
```

```
    avatar: 'xxxx.jpg',
```

```
  ])
```

```
},
```

```
  down: (queryInterface, Sequelize) => {
```

```
    /*
```

```
    Add reverting commands here.
```

```
    Return a promise to correctly handle asynchronicity.
```

```
    Example:
```

```
    return queryInterface.bulkDelete('Person', null, {});
```

```
  */
```

```
  queryInterface.bulkDelete('Users')
```

```
  }
```

```
};
```

utility 其实是 Egg 所依赖的东西，所以可以直接用，调用 MD5 进行密码验证，然后保存起来。MD5 加密的好处就是只要输入值是一样的，输出值就一样，避免了存储用户的明文密码，存储用户的明文密码容易给用户带来密码泄露风险，用户信任我们并使用我们的服务，我们就该对他们负责。

在 up 中定义了填错数据的逻辑，而在 down 中指定了回退的逻辑。

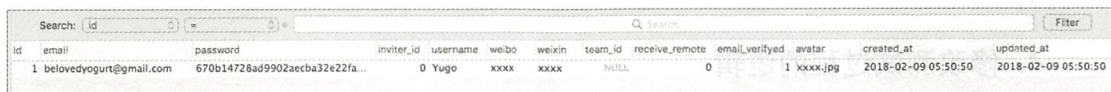
然后运行命令：

```
./node_modules/.bin/sequelize db:seed:all
```

假如成功，则通常会显示如下内容：

```
== 20180209035457-creat_user: migrating =====
== 20180209035457-creat_user: migrated (0.032s)
```

然后查看数据库中的数据，如图 3-1 所示。



id	email	password	inviter_id	username	weibo	weixin	team_id	receive_remote	email_verified	avatar	created_at	updated_at
1	belovedyogurt@gmail.com	670b14728ad9902aeba32e22fa...	0	Yugo	xxxx	xxxx	NULL	0	1	xxxx.jpg	2018-02-09 05:50:50	2018-02-09 05:50:50

图 3-1

当然填充数据并发一定要使用 seed，通过图形客户端添加也是一样的，只不过当数据量太大的时候不太方便。

3.2 发布一个插件

3.2.1 创建 Flash 插件

这一节我们制作一个 Flash 组件，并发布到 NPM 上。

1. 初始化项目

```
egg-init --type=plugin egg-msg-flash
```

egg-init 会帮我们初始化这个项目，其他配置项都以默认值为准。

2. 安装依赖

```
cd egg-msg-flash
```

```
npm install
```

3. 修改插件名字与依赖

修改 package.json 的 eggPlugin 字段内容，声明插件依赖 egg-session，这里的 Egg 开头不用写。

```
"eggPlugin": {
  "name": "flash",
  "dependencies": ["session"]
},
```

修改 config.default.js，key 为保存到 session 中的键。

```
exports.flash = {
  key: 'flash'
};
```

4. 修改测试过程的逻辑

安装依赖

```
cd test/fixtures/apps/msg-flash-test
npm link ../../../../
npm i egg-session
```

将 egg-msg-flash 软连接到测试项目，安装依赖 egg-session，因为 Flash 插件依赖 egg-session。

开启插件

修改测试项目的 config/plugin.js 开启插件，不配置参数，使用默认的即可。

```
exports.flash = {
  enable: true,
  package: 'egg-msg-flash',
};
```

修改 controller/home.js

因为前面修改了插件名字为 flash，所以在 home.js 中也要修改名字。

```
async index() {  
  this.ctx.body = 'hi, ' + this.app.plugins.flash.name  
}
```

修改 router.js

```
'use strict'  
  
module.exports = app => {  
  const { router, controller } = app  
  
  router.get('/', controller.home.index)  
  router.get('/session1', async (ctx, next) => {  
    ctx.flash_error({  
      ss: 'some error'  
    })  
    ctx.redirect('/session2')  
  })  
  router.get('/session2', async (ctx, next) => {  
    ctx.type = 'json'  
    ctx.body = ctx.flash  
  })  
  router.get('/session3', async (ctx, next) => {  
    ctx.flash = {  
      type: 'warning',  
      message: {  
        field: {  
          name: 'required'  
        }  
      }  
    }  
    ctx.request.flash('warning', {  
      field: {  
        name: 'required'  
      }  
    })  
    ctx.redirect('/session4')  
  })  
}
```



```
router.get('/session4', async (ctx, next) => {
  ctx.type = 'json'
  ctx.body = ctx.flash
})
}
```

在 `router.js` 中直接写执行的逻辑只是为了测试，在实际开发中，不要把逻辑写到 `router.js` 中。再修改一下 `test/msg-flash.test.js`，用来测试之前 `router.js` 中的逻辑：

```
it('should GET /', () => {
  return app
    .httpRequest()
    .get('/')
    .expect('hi, flash')
    .expect(200)
})

it('should GET /session1', () => {
  app
    .httpRequest()
    .get('/session1')
    .expect(302)
  app
    .httpRequest()
    .get('/session2')
    .expect(
      JSON.stringify({
        type: 'error',
        message: { ss: 'some error' }
      })
    )
})

it('should GET /session3', () => {
  app
    .httpRequest()
    .get('/session3')
    .expect(302)
```

```

app
  .httpRequest()
  .get('/session4')
  .expect(
    JSON.stringify({
      type: 'warning',
      message: {
        field: {
          name: 'required'
        }
      }
    })
  )
})

```

通过 `get` 方法可以访问应用的 URL，然后通过 `expect` 进行断言，传入的数字是断言状态码，字符串则说明是断言返回内容。

这里的测试不能跟随跳转，所以断言 302 之后，重新构建一个断言访问跳转之后的页面进行断言。

5. 测试一下

回到主项目下：

```
npm run test
```

目前我们什么代码都没写，所以出错是必然的，而且还有不少 `eslint` 错误。之所以有不少 `eslint` 错误，是因为 `npm run test` 命令中的 `eslint` 运行的是 `eslint . "--fix"`，这里多了一对双引号，所以并没有把 `--fix` 参数传递进去，我们可以执行一下：

```
./node_modules/.bin/eslint . -fix
```

这样可以解决一些小问题，但是还有一些其他约束需要修改，对于不太熟悉 `eslint` 的读者，其实关闭 `eslint` 就行，修改 `.eslintignore`：

```

coverage
node_modules
test
app

```

```
config
app.js
```

把这些目录都忽略了，这样就不会报错了。协作开发 `eslint` 确实不错，假如是独立开发者，只要代码风格简洁，则基本上不会有问题。如果代码写得冗余，则容易出问题。

6. 创建 `app/middleware/flash.js`

```
module.exports = ({ key }, app) => async (ctx, next) => {
  ctx.session[key] = ctx.session[key] || {}
  const flash = ctx.session[key]
  ctx.session[key] = {}

  function set(msg) {
    ctx.session[key] = msg
  }

  const get = () => flash

  Object.defineProperty(ctx, 'flash', {
    set,
    get,
    enumerable: true
  })

  ctx.request.flash = (type, msg) => {
    ctx.flash = { type, message: msg }
  }
  ;['success', 'error', 'info', 'warning'].forEach(type => {
    ctx['flash_' + type] = msg => (ctx.flash = { type, message: msg })
  })

  await next()

  if (ctx.status === 302 && ctx.session && !ctx.session[key]) {
    ctx.session[key] = flash
  }
}
```

然后在 `ctx.request` 中添加 `flash` 方法,这个方法用来支持 `passport` 的一些逻辑,`passport` 是一个封装了一些登录验证逻辑的插件,之后我们可能会用到。

新建 `app.js` 代码如下:

8. 查看代码测试覆盖率

运行上面的命令，我们可以看到如下所示的输出：

```

r - [Yugo nodelover] - [~/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-ms
g-flash] - [三 2 14, 11:44]
r - [$] <> npm run cov

> egg-msg-flash@1.0.0 cov /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter
5/5-2/egg-msg-flash
> egg-bin cov

test/msg-flash.test.js
  ✓ should GET /
  ✓ should GET /session1
  ✓ should GET /sessions

3 passing (7s)

===== Coverage summary =====
Statements : 78.26% ( 18/23 )
Branches   : 57.14% ( 4/7 )
Functions  : 50% ( 4/8 )
Lines      : 85% ( 17/20 )
=====

```




open coverage/lcov-report/index.html

打开 HTML 报告，如图 3-2、图 3-3 所示。

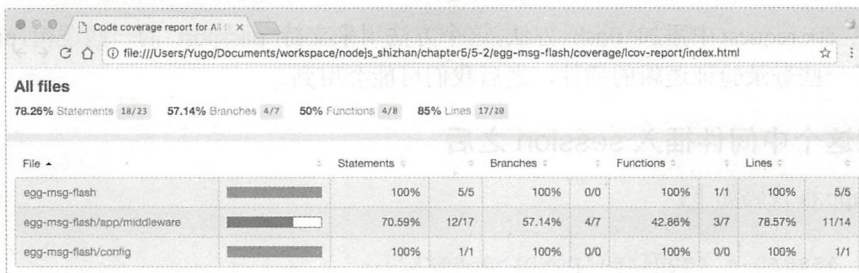


图 3-2

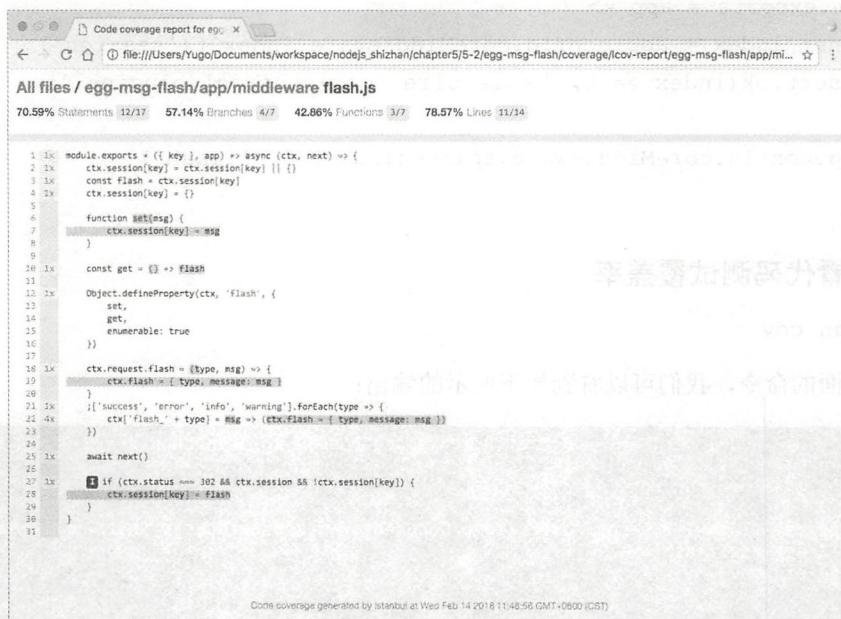


图 3-3

红色的部分提示测试没有覆盖到。其实我们在 router.js 中都用到过这些逻辑，只不过没有在 xx.test.js 中测试而已，因为中间件依赖一些运行环境，测试前需要 mock 一些变量，所以干脆忽略了。

我们可以将 `/* istanbul ignore file */` 加到中间件的文件的顶部。

于是覆盖率变成了 100%，如下所示。关于测试覆盖率更多的详细信息，可以阅读 <https://istanbul.js.org/docs/tutorials/mocha/> 上的内容。



```
> egg-msg-flash@1.0.0 cov /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter
5/5-2/egg-msg-flash
> egg-bin cov

test/msg-flash.test.js
  ✓ should GET / (58ms)
  ✓ should GET /session1
  ✓ should GET /session3

3 passing (18s)

===== Coverage summary =====
Statements : 100% ( 6/6 )
Branches   : 100% ( 0/0 )
Functions  : 100% ( 1/1 )
Lines      : 100% ( 6/6 )
=====
```

9. 更新 README.md

写好的东西就是为了给别人用的，所以我们要在 README.md 中告诉人家怎么用。

首先把所有的 Egg.js 替换为你的 GitHub 名字，稍后我们会生成一些小徽章来供项目显示。把 msgFlash 改为 flash，然后在 example 区块添加一下，代码如下：

```
ctx.flash = {
  type: 'success',
  message: {
    some: 'one'
  }
}

// or

ctx.flash_success({some: 'one'})

ctx.flash_error()
ctx.flash_info()
ctx.flash_warning()

ctx.request.flash(type, message)

// get flash by

ctx.flash
```



10. 提交到仓库

初始化：

```
git init
```

添加到暂存区：

```
git add .
```

到 <https://gitmoji.carloscuesta.me> 上选择一个你想要的图标，记住下面的 code，比如 `:tada:`，或者通过搜狗输入法输入 emoji 也可以，对于个人来说，没有具体标准，最好的标准就是没有标准，最好的代码就是没有代码，代码本身是很冰冷的，我们的工作就是让软件温暖人心。

```
git commit -m ':tada: init'
```

进入 GitHub 单击 new repository，创建一个代码仓库，如图 3-4 所示。

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: MiYogurt / Repository name: egg-msg-flash ✓

Great repository names are short and memorable. Need inspiration? How about laughing-goggles.

Description (optional):

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None Add a license: None ⓘ

Create repository

图 3-4



创建之后可以看到如图 3-5 所示的页面。

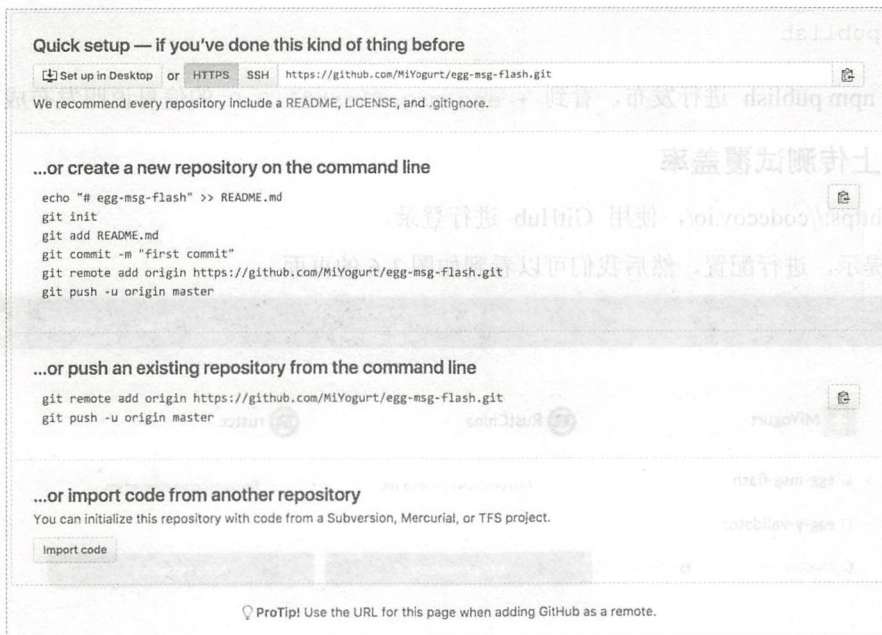


图 3-5

运行最后两条命令：

```
git remote add origin https://github.com/MiYogurt/egg-msg-flash.git
git push -u origin master
```

11. 发布到 npmjs

```
[Yugo: nodelover] - [~/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-ms
g-flash] - [三 2 14, 13:13]
└─ [$] <git:(master)> nrm use npm

Registry has been set to: https://registry.npmjs.org/

[Yugo: nodelover] - [~/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-ms
g-flash] - [三 2 14, 13:14]
└─ [$] <git:(master)> npm login
Username: miyogurt
Password:
Email: (this IS public) belovedyogurt@gmail.com
Logged in as miyogurt on https://registry.npmjs.org/.
```

```
nrm use npm
npm login
```

通过 `nrm use npm` 将源切换回 NPM，用 `npm login` 进行登录，如果没有账户，则要到



<https://www.npmjs.com> 上进行注册。

```
npm publish
```

通过 `npm publish` 进行发布，看到 `+ egg-msg-flash@1.0.0` 的信息说明发布成功。

12. 上传测试覆盖率

访问 <https://codecov.io/>，使用 GitHub 进行登录。

按照提示，进行配置，然后我们可以看到如图 3-6 的页面。

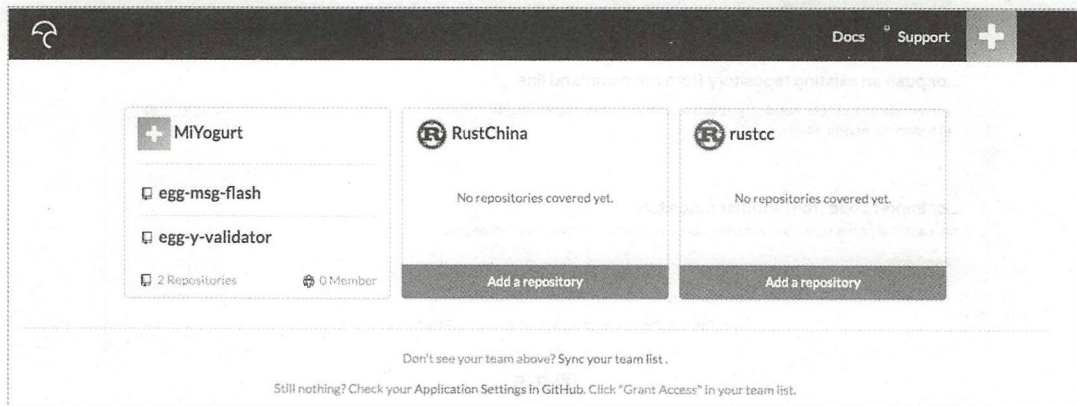


图 3-6

也有可能你的页面是空的，因为还没有提交仓库，稍后通过 CI 自动提交覆盖率测试报告，就会显示出来。

13. 测试 CI 是否成功

```
npm run ci
```

成功后如下所示。





```
[Yugo.nodelover] - [~/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-msg-flash] - [二 2 14, 13:16]
L - [?] <git:(master)> npm run ci

> egg-msg-flash@1.0.0 ci /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-msg-flash
> egg-bin pkgfiles --check && npm run lint && npm run cov

> egg-msg-flash@1.0.0 lint /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-msg-flash
> eslint .

> egg-msg-flash@1.0.0 cov /Users/Yugo/Documents/workspace/nodejs_shizhan/chapter5/5-2/egg-msg-flash
> egg-bin cov

test/msg-flash.test.js
  ✓ should GET / (156ms)
  ✓ should GET /session
  ✓ should GET /session1

3 passing (12s)

===== Coverage summary =====
Statements : 100% ( 6/6 )
Branches   : 100% ( 0/0 )
Functions  : 100% ( 1/1 )
Lines      : 100% ( 6/6 )
=====
```

在 <https://travis-ci.org> 上使用 GitHub 账户进行登录, 进入之后, 按照提示配置访问 GitHub 仓库的权限, 可以看到如图 3-7 所示的内容。

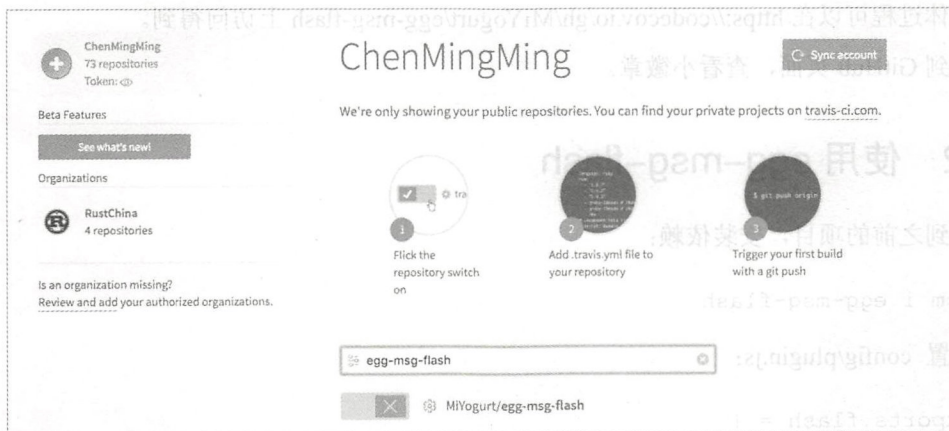


图 3-7

单击 sync account (同步仓库) 选项。在 filter 输入框中输入名字, 单击前面的 X 连接, 把它打开, 如图 3-8 所示。



图 3-8



单击 egg-msg-flash 连接，进入详情页面。

单击右上角的 more options 的 trigger build 选项，然后单击“确认”按钮，如图 3-9 所示。



图 3-9

现在是黄色的，说明还在构建测试中，等待变成绿色。

多刷新几次，发现小图标 build passing 已经变成绿色了，并且在 CI 过程中，也已经提交了代码测试覆盖率。

具体过程可以在 <https://codecov.io/gh/MiYogurt/egg-msg-flash> 上访问得到。

回到 GitHub 页面，查看小徽章。

3.2.2 使用 egg-msg-flash

回到之前的项目，安装依赖：

```
npm i egg-msg-flash
```

配置 config/plugin.js:

```
exports.flash = {  
  enable: true,  
  package: 'egg-msg-flash'  
}
```

配置 config/config.default.js:

```
config.flash = {  
  key: Symbol.for('flash')  
}
```




这里使用 Symbol 来作为 key，现在就完成安装了。

3.2.3 使用 egg-y-validator

这是验证表单的一个包，我们在 app/schemas 里定义的验证规则可以被加载，供 ctx.verify 调用。

安装：

```
npm i egg-y-validator
```

开启插件，配置 config/plugin.js:

```
exports.validator = {  
  enable: true,  
  package: 'egg-y-validator'  
}
```

配置 config/config.default.js:

```
config.validator = {  
  open: 'zh-CN',  
  languages: {  
    'zh-CN': {  
      required: '必须填 %s 字段'  
    }  
  },  
  async formatter(ctx, error) {  
    info('[egg-y-validator] -> %s', JSON.stringify(error, ' '))  
    throw new Error(error[0].message)  
  }  
}
```

3.3 规范化

在实际项目中，为了让大家学会 ESLint，笔者还是会开启 ESLint，不过我们要做一些配置，首先修改 package.json 中的 scripts，增加参数 fix 的支持。



3.3.1 添加新的 scripts 支持 ESLint 自修复

```
"scripts": {  
  "lint-fix": "eslint . --fix",  
  "test": "npm run lint-fix && npm run test-local"  
}
```

这里的 `fix` 参数是供我们在命令行里调用的，而不是给 VSCode 调用的，稍后我们会配置 VSCode，让编辑器帮我们格式化代码。

3.3.2 添加插件支持

1. 安装

笔者使用的是 VSCode，要开启 VSCode 对 ESLint 的支持，首先要全局安装 ESLint 命令行工具。

```
npm install -g eslint
```

然后安装 VSCode 的 ESLint 插件，单击“菜单”选项进行查看，然后单击“扩展”选项，输入 ESLint，单击“安装”按钮。

2. 如何解决错误

开启 ESLint 插件，当我们的语法不符合 ESLint 配置的时候，VSCode 就会抛出错误。

首先单击左下角的 × 号，如图 3-10 所示，然后打开错误窗口，错误输出如图 3-11 所示。

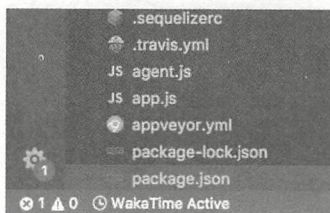


图 3-10

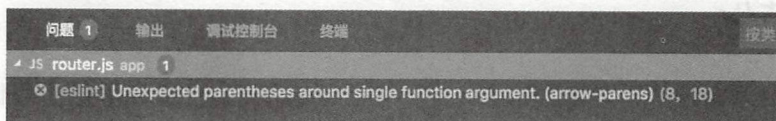


图 3-11

VSCode 告诉我们，router.js 的 8 行、18 列有问题，报错的意思大概就是，当箭头函数只有一个参数的时候，可以去掉这个参数的括号，而对应 ESLint 的规则是 arrow-parens。

假如不知道是什么规则报错，则可以输入错误中的一些关键字到 ESLint.cn 上搜索，得到相关信息。

既然报错了，那如何解决这个问题呢？第一种方式就是按照它的规格来修改代码。第二种方式就是通过 fix 参数来自动解决，但并不能解决所有问题。第三种方式是告诉 ESLint：我要禁用该规则。那么如何告诉 ESLint 呢？egg-init 帮我们初始化了 .eslintignore 和 .eslintrc 两个文件，.eslintignore 跟 .gitignore 类似，表示 ESLint 要忽略验证的文件，而 .eslintrc 则是 ESLint 的配置文件。

笔者的配置如下：

```
{
  "extends": "eslint-config-egg",
  "plugins": [
    "prettier"
  ],
  "rules": {
    "prettier/prettier": [
      "error"
    ],
    "space-before-function-paren": "off",
    "array-bracket-spacing": "off",
    "no-unused-vars": "off",
    "semi": "off",
    "comma-dangle": "off",
    "arrow-parens": "off"
  },
  "globals": {
    "use": true,
    "Controller": true
  }
}
```

extends 表示要继承的配置，eslint-config-egg 是 Egg 官方提供的配置，plugins 指定要添加的插件，rules 则是指定验证规则。这里我们把 arrow-parens 关掉，可以发现错误立刻就变成了 0。而 global 则是指定全局变量。知道这个技巧之后，基本上可以保证零错误，因为可以关

闭所有遇到的错误。

3.3.3 prettier 格式化工具

1. 安装依赖

prettier 与 VSCode 配合，可以实现保存代码文件立即格式化，但是保存后的格式同 ESLint 即 Egg 默认的规则有冲突，所以大家可以根据报错提示忽略一些错误。可以参考作者的 .eslintrc 上面的配置。

安装 prettier 的 ESLint 插件和 prettier:

```
npm install eslint-plugin-prettier prettier --save-dev
```

2. VSCode 的配置

```
"files.autoSave": "off",  
"eslint.autoFixOnSave": true
```

按 Cmd+, 可以“呼出”VSCode 的设置，开启 eslint.autoFixOnSave，保存代码之后会自动调用 eslint -fix 来解决错误，记得关闭自动保存，否则 Egg 在开发环境下会多次重启服务，得不偿失。

3. 创建.prettierrc

.prettierrc 会告诉 prettier 如何进行格式化，而对 VSCode 的配置不使用分号结尾，字符串使用单引号。

```
{  
  "semi": false,  
  "singleQuote": true  
}
```

3.3.4 同步代码编辑器配置

假如是合作开发，或者你有多台 PC，为了统一代码编辑器的配置，则需要安装 EditorConfig for VS Code，然后创建一个 .editorconfig，内容如下：

```
root = true
```

```
[*]
indent_style = space
indent_size = 2
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = false
insert_final_newline = true
```

配置生效如下：

- `[*]` 代表对于所有文件生效；
- `indent_style` 代表缩进是使用空格还是 `Tab` 控制符，这里是空格；
- `indent_size` 指缩进长度，这里是 2；
- `charset` 是文本以何种字符格式保存，这里是 UTF-8；
- `trim_trailing_whitespace` 指是否删除末尾的空格，这里是不删除；
- `insert_final_newline` 指是否自动插入末尾行，这里是自动插入。

关于 `editorconfig` 的更多信息可以查看 <http://editorconfig.org>，假如想在 VSCode 中快速生成，则可以安装 `editorconfiggenerator` 插件。

这样就可以在所有代码编辑器中同步一样的配置，基本上所有主流代码编辑器都有 `EditorConfig` 插件。

3.4 第一个 JSON 请求

3.4.1 给全局添加一些方法

1. 添加代码

新建 `init.js`，在其中添加一些全局方法，比如 `Controller`，这样我们创建控制器的时候，就不用每次都从 `Egg` 模块中导入。

把一些常用的无依赖的函数封装到全局是可行的，比如说调试函数、帮助函数，以及无副作用的函数。但是把所有的依赖都加载到全局中就不是一个正确的做法。而且把大对象加入全局对象上会发生内存泄漏，需要谨慎而行。

```
'use strict'
```



```

const path = require('path')
const R = require('ramda')

const check = R.curry((obj, key) => {
  if (typeof obj[key] === 'undefined') {
    return false
  }
  return true
})

const notInGlobal = key => !check({})(key)

function globalBaseInitial(baseDir) {
  const _use = dir => require(path.resolve(baseDir, dir))

  if (notInGlobal('check')) {
    global.check = check
  }

  if (notInGlobal('Controller')) {
    global.C = global.Controller = _use('app/controller/base')
  }

  if (notInGlobal('use')) {
    global.use = dir => {
      dir = dir.replace(/\.\/g, path.sep)
      return _use(dir)
    }
  }
}

module.exports = {
  globalBaseInitial
}

```

这里用到了 ramda，所以要安装 ramda 模块，因为 ramda 运行时有依赖，所以不指定参数，默认将其安装到运行依赖中。

```
npm install ramda
```

在后面的内容中我们会用到一些函数式编程，不过不是纯函数式编程，ramda 里有一些方法，可以代替 TryCatch、For、If 等，不过我们不会用这些，而是用一些帮助方法与函数式编程思维。

2. 函数柯里化

在项目下新建一个 test.js 来实践下面的代码：

```
const R = require('ramda')

const func = R.curry((a, b) => [a, b])

console.log(func(1, 2))
console.log(func(1)(2))
console.log(func(R.__, 2)(1))
```

运行结果如下：

```
[ 1, 2 ]
[ 1, 2 ]
[ 1, 2 ]
```

我们把一个匿名函数传给了 R.curry，这个匿名函数就是把两个参数按照顺序放到数组中，然后返回。把柯里化好的匿名函数赋值给 func，当调用柯里化函数时，一次性把参数传全，就可以直接调用函数，得出结果。假如只传递了一部分，那么它的返回值还是函数，还可以继续调用。也就是说继续传递参数，而且 ramda 还提供一个占位符函数 __，通过占位符，我们可以晚一些再传递。

为了更深入地了解函数柯里化，我们来实现一个简单的 curry 函数，不提供占位符功能。

```
function curry(fn) {
  const length = fn.length
  let args = []
  return (...arg) => {
    args = args.concat(arg)
    if (args.length === length) {
      return fn(...args)
    }
  }
}
```

```

    }
  }
}
const func2 = curry((a, b, c) => [a, b, c])

console.log(func(1, 2, 3))
console.log(func(1)(2)(3))

```

这里有两个要注意的要点：

- 通过访问函数的 `length` 属性可以获取它所需要传递参数的个数；
- 当所有参数都传递完毕，再调用 `fn`。

3. check 函数

```

const check = R.curry((obj, key) => {
  if (typeof obj[key] === 'undefined') {
    return false
  }
  return true
})

```

对 `check` 柯里化的好处就是，当我们要检测一个对象上的 `key` 的时候，我们可以先传递这个对象，然后把得到的函数保存起来，再进行多次调用。当然我们也可以手动柯里化代码，代码如下：

```

const check = (obj, key) => {
  if (typeof obj[key] === 'undefined') {
    return false
  }
  return true
}

const checkGlobal = key => check(global, key)

```

4. 函数组合

```

const notInGlobal = key => !check(global)(key)

```

这里我们先传递一个 `global` 对象，然后对返回值做 `not` 操作，因为我们要检测的对象是不存在的。这里用了一个简单的方式，其实还可以通过函数组合来实现。

```
const notInGlobal = key => !check(global)(key)
const notInGlobal = R.not(check(global))
const notInGlobal = R.compose(R.not, check(global))
```

可能大多数人都写过这样的代码：

```
somePromise.then(x => console.log(x))
```

其实它完全等于：

```
somePromise.then(console.log)
```

但是请注意这两个函数并不相等，因为 `!` 把后面的 `check(global)` 函数当成了 `function` 对象来解析，所以第二行的 `notInGlobal` 等于 `false`。

```
const notInGlobal = key => !check(global)(key)
const notInGlobal = !check(global)
```

第 2 行代码其实就是把 `!` 变成了函数，其实 `R.not` 类似于下面的代码：

```
const not = fn => (...args) => !fn(...args)
```

接收一个 `fn` 函数对这个 `fn` 的返回值取反，现在我们想象一下，假如在函数较多的情况。

```
a(b(c(d(e(f(g(1)))))))
```

一般函数式编程语言都会遇到这样的情况，这个嵌套写下去，让人头皮发麻，那么有没有什么办法来解决这个问题呢？答案是 `compose`，假如在 `clojure` 语言中那就是 `comp` 函数，`clojure` 是一门函数式编程语言，Java 与 `Lisp` 的结合体。

在说 `R.compose` 之前，我们看一下上面函数的调用顺序，`g→f→e→d→c→b→a`，是从右到左的。所以 `R.compose` 的调用方式也是从右到左的，以下三种方式等价。同时 `ramda` 提供从左到右的版本，它叫 `pipe` 函数。


```
const notInGlobal = R.not(check(global))
const notInGlobal = R.compose(R.not, check(global))
const notInGlobal = R.pipe(check(global), R.not)
```

接下来我们再实现一个 `compose` 函数。

```
const check = (obj, key) => {
  if (typeof obj[key] === 'undefined') {
    return false
  }
  return true
}

const compose = (...fns) => (...args) =>
  fns.reduceRight((acc, val) => val(acc), fns[fns.length - 1])(...args))

const composeTwoArgs = (a, b) => x => a(b(x))

const checkCurry = R.curry(check)

const fn = compose(R.not, checkCurry(global))

console.log(fn('use'))
console.log(global.use)

console.log(R.last([1, 2, 3]))
```

结果如下：

```
true
undefined
3
```

因为这里判断的是不存在，所以返回 `True` 是我们所期望的。假如只对两个函数进行组合，则可以使用 `composeTwoArgs` 函数，这只是特定的，而 `compose` 可以对任意个函数进行组合。

`compose` 必须是两层的函数，第一层用来传递连接的函数，第二层用来传递初始的值。当我们传递第一层函数的时候会得到一个 `fns`，它是一个函数的数组，因为 `compose` 是从右往左

的，所以我们应该从后面开始调用，恰好 ES6 提供了 `reduceRight` 函数，当然也可以先 `reverse` 反转顺序再 `reduce`。

为什么使用 `reduceXXX` 呢？因为 `reduceXXX` 是一个多变一的过程，并且这一次与上一次可以通过 `acc` 变量关联起来。假如我们对一个数组 `reduce` 求和，则输入为一个数组，数组中是数字，最后输出是一个求和后的数字。那么这就是一个数组，数组里是函数，最后的输出要求是所有函数调用的结果。在 `reduce` 求和的时候是对数组中的每一项与上一次求和的结果相加，在 `reduce` 函数的时候就是对数组中每一个函数进行调用，并把上一次调用的结果当作这次调用的参数，那么初始的值就为数组末尾函数调用的值，把 `compose` 第二层传递进来的参数值传给数组末尾的函数，把它的返回值作为初始值。

`fns[fns.length - 1]` 取最后一个函数，可以用 `R.last` 函数来取代。

5. 添加方法

```
function globalBaseInitial(baseDir) {
  const _use = dir => require(path.resolve(baseDir, dir))

  if (notInGlobal('check')) {
    global.check = check
  }

  if (notInGlobal('Controller')) {
    global.C = global.Controller = _use('app/controller/base')
  }

  if (notInGlobal('use')) {
    global.use = dir => {
      dir = dir.replace(/\.\/g, path.sep)
      return _use(dir)
    }
  }
}
```

在添加之前要做一下检测，`check` 方法就是刚刚的柯里化函数，`C` 和 `Controller` 就是 `app/controller/base.js` 里的控制器，假如有一些方法在所有控制器中都可以访问，而你又不想写到 `service` 里，那么就可以写到 `base.js` 里。而 `use` 方法其实就是添加一点乐趣，把 `require('app/contrller/home')` 变成了 `use('app.controller.home')`，`use` 添加了相对路径，即相对于

baseDir 目录的路径。

当添加完成之后,使用这些全局方法可能会出现验证错误,所以要在 ESLint 的 .eslintrc 里进行配置。

```
"globals": {  
  "use": true,  
  "Controller": true,  
  "C": true,  
  "check": true  
}
```

6. 装载辅助方法

在项目目录下创建 app.js 和 agent.js,添加如下代码:

```
'use strict'  
  
const { globalBaseInitial } = require('./init')  
  
globalBaseInitial(__dirname)  
  
module.exports = app => {}
```

3.4.2 全局化一些东西

1. 创建 BaseController

在 app/contrller 下新建 base.js,代码如下:

```
'use strict'  
  
const Controller = require('egg').Controller  
  
class BaseController extends Controller {}  
  
module.exports = BaseController
```

2. 测试 use

新建 app/schemas/signup/index.js:

```
'use strict'

module.exports = {
  name: {
    type: 'string',
    required: true
  }
}
```

修改 app/controller/home.js:

```
'use strict'

class HomeController extends Controller {
  async index() {
    console.log(global.use)
    const r = use('app.schemas.signup')
    this.ctx.type = 'json'
    this.ctx.body = JSON.stringify(r)
  }
}
```

```
module.exports = HomeController
```

通过 `npm run dev` 启动之后查看主页输出, 可以看到如图 3-12 所示的输出结果。

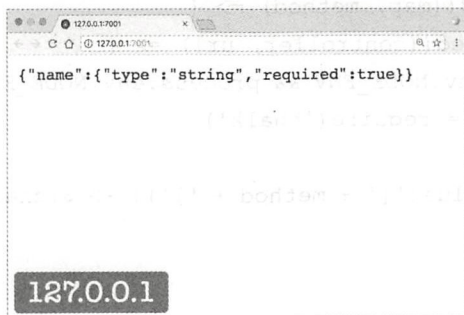


图 3-12

3.4.3 自动路由

1. 新建 app/api.js

```
'use strict'

module.exports = ctx => ({
  post: {
    '/signup': ctx.user.signup, // 注册
    '/signin': ctx.user.signin // 登录
  }
})
```

对于前端的 API，我们希望单独拿出来一个文件，写到 `api.js` 里。结构如上面代码所示，我们希望把相同的请求方法放在一个对象里，然后把 URL 作为 key，控制器作为 value。这是一个函数，接收一个 `controllers` 对象，因为控制要从这个对象获取，相当于通过函数保存 `controllers` 的作用域。当然也可以把字符串类型作为 value，然后在下面的 `utils.js` 中，将字符串处理为控制器函数。接下来我们要编写 `utils.js`，实现自动化添加路由。

2. 新建 app/utils.js

先安装依赖，安装到开发依赖，`chalk` 可以提供控制台彩色的输出。

```
npm i chalk -D
'use strict'
const { forEachObjIndexed } = require('ramda')

function initRouterMap(prefix, maps, router) {
  forEachObjIndexed((map, method) => {
    forEachObjIndexed((controller, url) => {
      if (process.env.NODE_ENV !== 'production') {
        const chalk = require('chalk')
        console.log(
          `${chalk.blue('[' + method + ']}') -> ${chalk.red(prefix + url)}`
        )
      }

      router[method](prefix + url, controller)
    })
  })
}
```

```

    }, map)
  }, maps)
}

module.exports = {
  initRouterMap
}

```

ramda 提供的 `forEachObjIndexed` 方法可以遍历对象，`initRouterMap` 接收一个 `prifix`，因为 API 相关的都在 `/api/v1` 下，`v1` 是版本号，当我们升级到 `v2` 版本的时候，不会造成路由冲突。然后就是遍历 `maps` 上的配置，调用对应的 `method` 添加相应的 URL，即与之对应的控制器，并且在开发环境中输出路由表。

在控制台可以看到以下输出：

```

[post] -> /api/v1/signup
[post] -> /api/v1/signin

```

在 `router.js` 中调用它们：

```

'use strict'

const init = require('./util').initRouterMap

/**
 * @param {Egg.Application} app - egg application
 */
module.exports = app => {
  const { router, controller } = app
  router.get('/', controller.home.index)
  init('/api/v1', require('./api')(controller), router)
}

```

3. 新建 `app/contrller/user.js`

```
'use strict'
```

```
class User extends C {
```

```
  /**
```

```
* @description 注册
* @memberof User
*/
async signup() {
  this.ctx.body = '注册'
}
/**
* @description 登录
*
* @memberof User
*/
async signin() {
  this.ctx.body = '登录'
}
}

module.exports = User
```

这里没有使用装饰器来添加路由，是因为装饰器需要 `babel`，当前的 Node 版本基本上都支持 `babel`，想要开启 `import` 语法，则在运行时指定 `experimental-modules` 参数即可。一些 `core.js` 中的库没有引入，所以就没有 `Reflect` 等对象，这要看后期要不要用，如果用就导入这些库。

3.4.4 创建 PostMan 测试

1. 安装 PostMan

对于 API，我们可以在 `test` 里写测试，也可以用图形化界面来实现，这里我们用图形化界面实现，首先安装 Postman，在 <https://www.getpostman.com/> 上可以下载获取。

单击左上角的 `new` 按钮，然后单击 `Collection` 选项，创建一个组，输入名字为 `miao`。

选择 `Post` 方法，输入 `http://localhost:7001/api/v1/signin`，单击 `send` 按钮发起请求，这时候会得到一个 `403` 状态的返回值，因为我们没有配置 `CSRF`。暂时不管成功与否，单击 `save` 按钮，选择 `miao collection` 选项，注释写“登录”，然后保存。

2. 解决 CSRF 问题

`CSRF` 是一种机制，确保用户是在本站发起的请求，假如银行提供一个加钱的 API 接口，

那岂不是谁都可以给自己卡里加钱？所以通常会有一个 Token（随机字符串），它记录在你的 Session 或 Cookie 中，校验上一次访问生成的 Token 与这一次是否相等。当然银行的 API 还有更多的安全策略，甚至不可能存在这种加钱 API。

CSRF 是由 egg-security 插件提供的，它其实是中间件，下面把它的核心源码列出来，第一个功能是生成 CSRF，第二个功能是验证，它们都在 ctx 对象上。

```
ensureCsrfSecret(rotate) {
  if (this[CSRF_SECRET] && !rotate) return;
  debug('ensure csrf secret, exists: %s, rotate: %s', this[CSRF_SECRET],
    rotate);
  const secret = tokens.secretSync();
  this[NEW_CSRF_SECRET] = secret;
  const { useSession, sessionName, cookieDomain, cookieName } =
    this.app.config.security.csrf;

  if (useSession) {
    this.session[sessionName] = secret;
  } else {
    const cookieOpts = {
      domain: cookieDomain && cookieDomain(this),
      signed: false,
      httpOnly: false,
      overwrite: true,
    };
    this.cookies.set(cookieName, secret, cookieOpts);
  }
},

assertCsrf() {
  if (utils.checkIfIgnore(this.app.config.security.csrf, this)) {
    debug('%s, ignore by csrf options', this.path);
    return;
  }

  if (!this[CSRF_SECRET]) {
    debug('missing csrf token');
    this[LOG_CSRF_NOTICE]('missing csrf token');
    this.throw(403, 'missing csrf token');
```



```
    }  
    const token = this[INPUT_TOKEN];  
  
    // AJAX requests get csrf token from cookie, in this situation token will  
    equal to secret  
    // synchronize form requests' token always changing to protect against  
    BREACH attacks  
    if (token !== this[CSRF_SECRET] && !tokens.verify(this[CSRF_SECRET],  
    token)) {  
      debug('verify secret and token error');  
      this[LOG_CSRF_NOTICE]('invalid csrf token');  
      this.throw(403, 'invalid csrf token');  
    }  
  },  
},
```

中间件代码如下：

```
const debug = require('debug')('egg-security:csrf');  
const typeis = require('type-is');  
const utils = require('../utils');  
  
module.exports = options => {  
  return function csrf(ctx, next) {  
    if (utils.checkIfIgnore(options, ctx)) {  
      return next();  
    }  
  
    // ensure csrf token exists  
    ctx.ensureCsrfSecret();  
  
    // ignore requests: get, head, options and trace  
    const method = ctx.method;  
    if (method === 'GET' ||  
      method === 'HEAD' ||  
      method === 'OPTIONS' ||  
      method === 'TRACE') {  
      return next();  
    }  
  }  
}
```

```

if (options.ignoreJSON && type.is(ctx.get('content-type'), 'json')) {
  return next();
}

const body = ctx.request.body || {};
debug('%s %s, got %j', ctx.method, ctx.url, body);
ctx.assertCsrf();
return next();
};
};

```

我们发现有一个 `options.ignoreJSON` 的配置项，可以配置一下，还可以使用通用配置项的 `ignore`，文档在 <https://eggjs.org/zh-cn/basics/middleware.html#通用配置> 中可以找到。

修改 `config.default.js`:

```

config.security = {
  csrf: {
    ignoreJSON: true
  }
}

```

开启之后，从 `body` 里找到 `raw` 选项，把后面的参数改成 `JSON`，不要在 `body` 里选择 `form-data` 或者 `x-www-form-urlencoded` 再去设置 `Header` 头，这样是不行的，直接改成 `raw` 后选择 `JSON` 即可。

3.5 注册服务

3.5.1 Invitation 模型

首先给 `Invitation` 添加生成邀请码的逻辑，在这里笔者会介绍一些注释语法，在个人项目中哪怕是一个人维护也要写一些注释，只不过这个注释更多是一些需要完成的任务。

1. 安装依赖

```
npm install uuid
```

2. 添加逻辑

给 `app/model/invitation.js` 添加方法：

```
const uuid = require('uuid/v1')
/**
 * * 检验邀请码是否有效
 * @param {string} code 邀请码
 * @return {Promise<boolean>} 是否有效
 */
Invitation.exits = async code => {
  const instance = await Invitation.findOne({
    where: {
      code
    }
  })
  return Boolean(instance)
}

// * 生成邀请码 Hook
// ? FIXME: 待优化
Invitation.beforeCreate(async (instance, options) => {
  const generatorCode = async code => {
    code = uuid().split('-')[0]
    if (await Invitation.exits(code)) {
      return await generatorCode()
    }
    return code
  }
  if (!instance.code) {
    instance.code = await generatorCode()
  }
})
```

- `exits` 方法接收一个邀请码，会检测该邀请码是否有效。
- `Invitation.beforeCreate` 可以为 `Invitation` 添加一个生命周期函数钩子，在创建之前执行其中的逻辑，通过 `uuid` 生成随机码，假如存在该邀请码就重新生成。

生命周期大致有如下几个阶段：

```
(1)
  beforeBulkCreate(instances, options)
  beforeBulkDestroy(options)
  beforeBulkUpdate(options)
(2)
  beforeValidate(instance, options)
(-)
  validate
(3)
  afterValidate(instance, options)
  - or -
  validationFailed(instance, options, error)
(4)
  beforeCreate(instance, options)
  beforeDestroy(instance, options)
  beforeUpdate(instance, options)
  beforeSave(instance, options)
  beforeUpsert(values, options)
(-)
  create
  destroy
  update
(5)
  afterCreate(instance, options)
  afterDestroy(instance, options)
  afterUpdate(instance, options)
  afterSave(instance, options)
  afterUpsert(created, options)
(6)
  afterBulkCreate(instances, options)
  afterBulkDestroy(options)
  afterBulkUpdate(options)
```

3.5.2 注释

1. 安装插件

首先使用 VSCode 安装几个关于注释的插件。

- Better Comments, 给注释上色;
- Document This, 自动生成注释;
- TODO Highlight, 高亮 TODO, 并搜寻所有 TODO。

打开 VSCode 开启 TODO Highlight 路径显示:

```
"todohighlight.toggleURI": true,
```

以下是显示效果:

```
/**
 * * 检验邀请码是否有效
 * @param {string} code 邀请码
 * @return {Promise<boolean>} 是否有效
 */
Invitation.exits = async code => {
  const instance = await Invitation.findOne({
    where: {
      code
    }
  })
  return Boolean(instance)
}

// * 生成邀请码 Hook
// ? 待优化
Invitation.beforeCreate(async (instance, options) => {
  const generatorCode = async code => {
    code = uuid().split('-')[0]
    if (await Invitation.exits(code)) {
      return await generatorCode()
    }
    return code
  }
  if (!instance.code) {
    instance.code = await generatorCode()
  }
}

return Invitation
}
```

```
// ! error
// ? info
// * success
```

第一行会变成红色，第二行会变成蓝色，第三行会变成绿色，这是由 Better Comments 插件提供的，虽然它也支持 Todo 高亮，但是它不支持扫描，所以又安装了 TODO Highlight。这两个插件都可以自定义高亮的关键字与颜色，它们都可以在插件的配置文档里找到。

TODO Highlight 只提供两个关键字 TODO 和 FIXME，TODO 代表这个地方还有事情没有做完。在开发过程中，不可能一次性就把所有的事情都做完，所以我们用 TODO 来做记录，而 FIXME 则代表这个地方有问题，甚至无法运行，等待有人去解决。

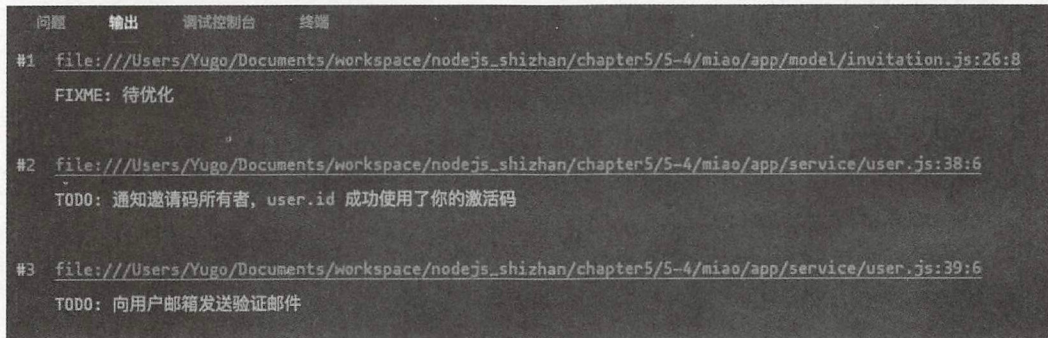
2. 查找 TODO

按 CMD + Shift + P 组合键，打开命令面板，选择 List highlighted annotations，然后选择 ALL。



```
>todo|
TODO-Highlight: List highlighted annotations
Todo: Open
TODO-Highlight: Toggle highlight
```

我们就可以看到如下输出，按住 option 然后单击这些路径，就可以快速到达这个位置，记得一定要把之前提到的配置打开，否则文件路径会以 # 结尾，那么就无法跳转。



```
问题 输出 调试控制台 终端
#1 file:///Users/Yugo/Documents/workspace/nodejs_shizhan/chapter5/5-4/miao/app/model/invitation.js:26:8
  FIXME: 待优化

#2 file:///Users/Yugo/Documents/workspace/nodejs_shizhan/chapter5/5-4/miao/app/service/user.js:38:6
  TODO: 通知邀请码所有者, user.id 成功使用了你的激活码

#3 file:///Users/Yugo/Documents/workspace/nodejs_shizhan/chapter5/5-4/miao/app/service/user.js:39:6
  TODO: 向用户邮箱发送验证邮件
```

当然 TODO 也可以写到我们之前的甘特图中，但是这样就要记录是哪个文件、在哪一行，比较麻烦，所以记录到编辑器中是最好的，但是千万别忘记更新甘特图。

3. 方法注释

```
/**
 * * 哈希密码 Hooks
 * @param {User} instance 用户实例
 * @return {void}
 */
```

在下面的代码中，你可能会看到类似这样的注释，第一行其实是简介，对方法进行一个简单的说明，也可以用 @desc 来注释，不过笔者喜欢给它一个颜色，所以多了一个 **，因为

没必要使用 JSDoc 或者 ESDoc 去生成一些 API 文档。这不是一个给别人提供代码调用的包，而是一个 Web 应用，而且并不是开源的，所以没必要按照 JSDoc 或者 ESDoc 的标准来写注释，只要能理解就行，格式如下：

@类型 {参数类型} 参数名称 参数简介

常用的有 @param 和 @return。

关于更多内容，可以查看相关的 Tag：

- JSDoc, <http://usejsdoc.org/>。
- ESDoc, <https://esdoc.org/manual/tags.html>。

3.5.3 User 模型

1. 安装依赖

```
npm install bcrypt
```

加密码更安全的方式是使用 bcrypt 算法。

2. 添加逻辑

```
/**
 * * 哈希密码 Hooks
 * @param {User} user 用户实例
 * @return {void}
 */
```

```
async function hashPwd(user) {
  if (!user.changed('password')) {
    return
  }
  user.password = await bcrypt.hash(user.password, 10)
}
```

```
User.beforeSave(hashPwd)
```

```
/**
 * * 用户登录方法
 * @param {string} email 邮箱
```

```

* @param {string} password 密码
* @return {(User|boolean)} 登录成功的用户
*/
User.Auth = async function (email, password) {
  const user = await this.findOne({
    where: {
      email
    }
  })
  if (await bcrypt.compare(password, user.password)) {
    return user
  }
  return false
}

```

为了让 User 模型自动 Hash 密码，可以为模型添加一个 beforeSave 的钩子，当用户修改 password 的时候，就通过 bcrypt 库 Hash 一下密码。

再添加一个用户验证的方法，接收邮件与密码，且 auth 是 user 的静态方法。

3.5.4 修改控制器

给 user 控制器添加 signUp 方法，记得把路由的参数也修改一下。

```

/**
* @description 注册
* @member User
*/
async signUp() {
  await this.ctx.verify('user.signup', 'body')
  const json = await this.ctx.service.user.signUp()
  this.ctx.body = json
}

```

调用 user.signup 来验证逻辑，再调用 user 服务的 signUp 方法，然后通过 JSON 将它的结果返回 api.js。

```

module.exports = ctx => ({

```



```

post: {
  '/signup': ctl.user.signup, // 注册
  '/signin': ctl.user.signin // 登录
}
})

```

3.5.5 添加验证逻辑

app/schemas/user/signup.yaml:

```

username:
  required: true
  message: '请填写用户名'
password:
  - min: 6
    message: '密码长度不能小于 6 位'
  - required: true
    message: '请填写密码'
confirm_password:
  - required: true
    message: '请填写确认密码'
  - validator: !!js/function >
      function validator(ctx) {
        return async function (rule, value, callback, source, options) {
          if(ctx.request.body.password === value){
            return callback();
          }
          callback([ { message: '密码与确认密码不匹配', field: 'confirm_password' } ])
        }
      }
email:
  type: 'email'
  required: true
  message: '用户邮箱有误'
code:
  - required: true
    message: '必须填写邀请码'
  # - validator: !!js/function >

```

```

#   function validator(ctx) {
#       return async function (_, value, cb) {
#           const invitation=await ctx.app.model.Invitation.find({where:{code:value}})
#           if (!invitation) {
#               return cb([{ message:'没有找到该邀请码', field:'code'}])
#           }
#           return cb()
#       }
#   }

```

egg-y-validator 经过笔者的修改之后, yaml 可以支持 function, 通过 function 可以自定义验证, 由于需要获取 ctx 做数据查询, 利用函数的作用域保存 ctx 的原理, 这个方法是一个二层函数, 从第一层获取 ctx 对象, 删除 code 的自定义验证逻辑, 查询出来的 invitation 还需要使用, 而在这里查询不容易传递 invitation, 所以把邀请码有效性的逻辑移到 service 里。

同时也可以使用 JS 文件, egg-y-validator 也支持 JS 文件和 TOML 文件的载入, 更多内容可以在 README.md 中找到。而验证的规则是使用 async-validator 库。

简单地叙述一下验证规则:

- required, 表示该字段是必需的;
- type, 表示该字段的类型, 默认是 string;
- message, 该字段验证出错的时候提示的消息。

假如抛出了错误, 则会在 egg-y-validator 的 config 的 formatter 中进行错误处理, 这里给的是 400 错误, 也就是说在前端通过状态码来判断是否有错误。

假如不懂 yaml 和 toml, 则可以到 <https://nodelover.me/course/config-file> 上观看笔者录制的关于这两种配置文件的入门视频, 或者阅读官方英文文档。

通常, 控制器只能用作跳转和验证, 其他跟 ORM 有关的放在 service 里, 可复用性会更高。

3.5.6 帮助方法

新建 app/extend/helper.js:

```
'use strict'
```

```
const R = require('ramda')
```

```
module.exports = {
  where(obj, ...args) {
    return Object.assign(
      {
        where: obj
      },
      ...args
    )
  },
  throw(code, field, message) {
    this.ctx.status = code
    throw [{ field, message }]
  },
  range(start, end) {
    const _range = function* name(start, end) {
      let index = start
      if (typeof end === 'undefined') {
        end = start
        index = 0
      }
      while (index < end) {
        yield index++
      }
    }
    return Array.from(_range(start, end))
  }
}
```

- `where` 用于帮助构建 ORM 查询，因为每次都要执行 `{where: { name: 'bob' }}`，现在只需要执行 `where({name: 'bob'})`；
- `throw` 用于抛出跟验证错误相同的格式；
- `range` 用于快速生成一个 `start` 到 `end-1` 的数组。

`Array.from` 接收一个 `ArrayLike` 或者迭代器，`range` 也可以用更简单的版本，拥有 `length` 属性，可以看作 `ArrayLike`。代码如下：

```
Array.from({length: end - start}, (_, k) => start + k)
```

3.5.7 User 服务

```
'use strict'
const R = require('ramda')
class User extends S {
  constructor(ctx) {
    super(ctx)
    this._User = this.ctx.model.User
    this._Invitation = this.ctx.model.Invitation
    this.where = this.ctx.helper.where
  }
  /**
   * * 校验邀请码有效性
   * @param {string} code 邀请码
   * @return {boolean} 是否有效
   * @memberof User
   */
  async checkInvitation(code) {
    const invitation = await this._Invitation.find(this.where({ code }))
    if (!invitation || invitation.use_user_id) {
      return this.ctx.helper.throw(400, 'code', '无效的邀请码')
    }
    return invitation
  }
  /**
   * * 生成邀请码
   * @param {number} user_id 用户 ID
   * @param {number} length 生成的个数
   * @return {Array<Invitation>} 所生成的邀请码数组
   * @memberof User
   */
  async generatorInvitation(user_id, length) {
    const invitation_promises = this.ctx.helper.range(length).map(() => {
      return this._Invitation.create({ user_id })
    })
    return Promise.all(invitation_promises)
  }
}
```



```

* * 注册逻辑
* TODO: 通知邀请码所有者, user.id 成功使用了你的激活码
* TODO: 向用户邮箱发送验证邮件
* @return {Object} 注册成功的用户与生成的邀请码
* @memberof {User}
*/
async signUp() {
  const body = this.ctx.request.body
  const invitation = await this.checkInvitation(body.code)
  const user = await this._User.create(
    R.pick(['username', 'password', 'email'], body)
  )
  /* eslint-disable no-proto */
  console.dir(user.__proto__)
  console.dir(invitation.__proto__)
  invitation.use_user_id = user.id
  invitation.use_username = user.username
  await invitation.save()
  const invitations = await this.generatorInvitation(user.id, 5)
  return { user, invitations }
}
}

module.exports = User

```

在 `constructor` 注入一些方法到 `this` 中, 供我们在方法里调用。

在触发 `signUp` 的时候, 会先去校验邀请码的有效性, 假如如有误, 则抛出跟验证错误相同的格式。当校验通过时创建用户, 并把该邀请码的使用者的信息记录到邀请码表中。

`User` 跟 `Invitation` 模型建立了很多关系, 而且会往它们各自的实例上挂载一些方法, 通常这些方法的名称并不容易记忆, 可以通过打印它们的原型链得到, 代码如下:

```

/* eslint-disable no-proto */
console.dir(user.__proto__)
console.dir(invitation.__proto__)

```

这里会遇到 ESLint 错误, 可以通过注释临时关闭来解决。因为用到了反范式设计, 多了一个 `use_username` 字段, 所以直接赋值修改更方便, 通过 `User` 与 `Invitation` 自动生成的



`User.setMy_used_invitaion` 只能设置 `invitation.use_user_id = user.id`，之所以是 `setMy_used_invitaion`，是因为在建立关系的时候用的 `as` 是 `my_used_invitaion`，假如觉得难看，则可以把这种下画线分割方式改成驼峰规则。

笔者的个人风格是对实例方法会使用小驼峰规则，静态方法使用大驼峰规则，文件名和变量使用下画线分割，跟 `Vue` 的风格比较相似。

```
R.pick(['username', 'password', 'email'], body)
```

等价于以下逻辑：

```
{
  username: body.username,
  password: body.password,
  email: body.email
}
```

对于 `generatorInvitation` 方法，因为要一次性生成 5 个验证码，我们可以先构建 5 个 `Promise`，然后通过 `Promise.all` 创建 5 个邀请码。

还有一些逻辑需要完善，但是此刻没办法完成，所以写到 `Todo` 中。

3.5.8 PostMan 测试

先用 `Sequel Pro` 图形界面创建一个邀请码数据，比如 123456，如图 3-13 所示。

id	code	user_id	use_user_id	use_username	created_at	updated_at
1	123456	NULL	NULL		2018-02-18 16:23:28	2018-02-19 05:54:54

图 3-13

向 `localhost:7001/api/v1/signup` 路由提交请求，数据如图 3-14 所示。

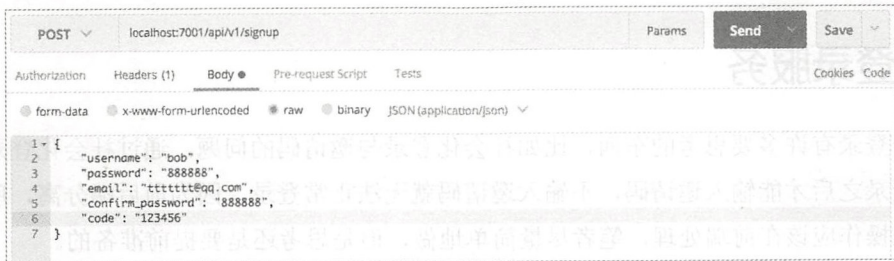


图 3-14



```
{
  "username": "bob",
  "password": "888888",
  "email": "ttttttt@qq.com",
  "confirm_password": "888888",
  "code": "123456"
}
```

得到如图 3-15 所示的结果，表明我们的用户与邀请码创建成功了。

```
1- {
2-   "user": {
3-     "receive_remote": 0,
4-     "email_verified": 0,
5-     "id": 30,
6-     "username": "bob",
7-     "password": "$2a$10$3/g3XLEVwzUpIvWFRbyLG0bxTP0hOeI4CHG1cB38oc36e.SZbxre",
8-     "email": "ttttttt@qq.com",
9-     "updated_at": "2018-02-20T06:10:13.749Z",
10-    "created_at": "2018-02-20T06:10:13.749Z"
11-  },
12-  "invitations": [
13-    {
14-      "id": 67,
15-      "user_id": 30,
16-      "updated_at": "2018-02-20T06:10:13.956Z",
17-      "created_at": "2018-02-20T06:10:13.956Z",
18-      "code": "b6ef0c80"
19-    },
20-    {
21-      "id": 68,
22-      "user_id": 30,
23-      "updated_at": "2018-02-20T06:10:13.956Z",
24-      "created_at": "2018-02-20T06:10:13.956Z",
25-      "code": "b6ef3390"
26-    },
27-    {
28-      "id": 69,
29-      "user_id": 30,
30-      "updated_at": "2018-02-20T06:10:13.956Z",
31-      "created_at": "2018-02-20T06:10:13.956Z",
32-      "code": "b6ef3391"
33-    },
34-    {
35-      "id": 70,
36-      "user_id": 30,
37-      "updated_at": "2018-02-20T06:10:13.957Z",
38-      "created_at": "2018-02-20T06:10:13.957Z",
39-      "code": "b6ef3392"
40-    }
41-  ]
42-}
```

图 3-15

还可以提交一些不符合规定的数据试一试。

3.6 登录服务

对于登录有许多要思考的东西，比如社会化登录与邀请码的问题，通过社会化登录，只有在用户登录之后才能输入邀请码，不输入邀请码就无法正常登录，因为前后端分离，所以登录之后无法操作应该在前端处理，笔者尽量简单地做，但是思考还是要提前准备的。

图 3-16



1. 安装依赖

```
npm install egg-passport egg-passport-local
npm install egg-jwt
```

passport 中封装了登录的公用组件，而 passport-local 则是本地登录的逻辑。在开发的时候往 passport-local 里传入一个用于验证用户名与密码的回调，passport 会帮我们 from ctx 上获取用户名与密码，传入我们在 passport-local 的回调中，供我们验证。

passport 通常会把用户登录的信息记录到 Session 里，而前后端分离则是通过一个加密的 Token 来验证用户的身份，假如一定要开启 Session，那么在前端提交请求的时候一定要带上 Cookie，因为 session_id 是存储在 Cookie 中的。

2. 添加环境变量判断

```
if (notInGlobal('DEV')) {
  global.DEV = process.env.NODE_ENV !== 'production'
}
```

在 init.js 中添加一些变量，为了便于调试，要有一个环境变量，别忘记在 .eslintrc 中添加 global 变量。

3. 开启插件

```
exports.passport = {
  enable: true,
  package: 'egg-passport'
}

exports.jwt = {
  enable: true,
  package: 'egg-jwt'
}

exports.passportLocal = {
  enable: true,
  package: 'egg-passport-local'
}
```

配置插件：

```
const R = require('ramda')
```




```
const chalk = require('chalk')
config.jwt = {
  secret: '123456',
  enable: true,
  ignore(ctx) {
    const paths = ['/api/v1/signin', '/api/v1/signup']
    if (DEV) {
      const tip = `${chalk.yellow('[ JWT ]')} --> ${
        R.contains(ctx.path, paths)
          ? chalk.green(ctx.path)
          : chalk.red(ctx.path)
      }`
      console.log(tip)
    }
    return R.contains(ctx.path, paths)
  }
}

exports.passportLocal = {
  usernameField: 'email',
  passwordField: 'password'
}
```

对于登录与注册的 URL 我们不要求 JWT 验证，通过 `ignore` 函数返回 `boolean` 可以判断是否跳过了验证，`secret` 则是加密的原始密码。

因为使用 `email` 和 `password` 登录，所以将原来的 `username` 改成 `email`。

```
function wrapMiddleware(mw, options) {
  // support options.enable
  if (options.enable === false) return null;

  // support generator function
  mw = utils.middleware(mw);

  // support options.match and options.ignore
  if (!options.match && !options.ignore) return mw;
  const match = pathMatching(options);

  const fn = (ctx, next) => {
    if (!match(ctx)) return next();
    return mw(ctx, next);
  }
```



```

};
fn._name = mw._name + 'middlewareWrapper';
return fn;
}

```

在 egg-core 的中间件载入的过程中,会对中间件进行 wrapMiddleware,这个 wrapMiddleware 就会添加通用选项,比如 enable、match 和 ignore。wrapMiddleware 其实跟装饰器差不多,也就是在执行该中间件之前,添加一些验证逻辑。

4. 自动函数

修改 app/util.js 为以下逻辑:

```

'use strict'
const { forEachObjIndexed, forEach } = require('ramda')
const chalk = require('chalk')
function initRouterMap(prefix, maps, router) {
  forEachObjIndexed((map, method) => {
    forEachObjIndexed((controller, url) => {
      if (DEV) {
        console.log(
          `${chalk.blue('[ ' + method + ' ]')}` -> `${chalk.red(prefix + url)}`
        )
      }
      router[method](prefix + url, controller)
    }, map)
  }, maps)
}

function mountPassportToController(keys, passport, controller) {
  if (!controller.passport) {
    controller.passport = {}
  }
  forEach(value => {
    if (DEV) {
      console.log(`${chalk.blue('[ mount passport ]')}` ${chalk.red(value)}`)
    }
    controller.passport[value] = passport.authenticate(value, {
      session: false,

```





```

    successRedirect: undefined
    // @see https://github.com/eggjs/egg-passport/blob/
    0afce5b0d5fbc107730e10dad6aff915c03cf079/lib/passport.js#L41
  })
  }, keys)
}

function installPassport(passport, { verify }) {
  passport.verify(verify)
}

module.exports = {
  initRouterMap,
  mountPassportToController,
  installPassport
}

```

- `mountPassportToController` 将 `passport` 生成的中间件挂载到控制器上，调用 `passport.authenticate` 方法获得中间件，第一个参数代表验证逻辑，比如 `local`、`weibo` 和 `github`，分别对应 `egg-passport-local`、`egg-passport-weibo` 和 `egg-passport-github` 插件，第二个参数是配置选项；
- `installPassport` 是给 `passport` 添加如何验证的逻辑。

之所以要配置 `successRedirect` 为 `undefined`，是因为我们要直接返回一个 `Token`，不需要成功重定向，而 `egg-passport` 添加了默认选项，也就是以下这段逻辑，设置 `successRedirect` 为 `undefined` 则可以跳过以下逻辑：

```

if (!options.hasOwnProperty('successRedirect') && !options.hasOwnProperty(
  'successReturnToOrRedirect')) {
  options.successReturnToOrRedirect = '/';
}

```

通过 VSCode 的搜索功能输入 `authenticate` 就可以找到该代码。它在 `passport.js` 下面。

5. 路由

修改 `app/router.js` 代码：

```

const init = require('./util').initRouterMap
const mount = require('./util').mountPassportToController

```





```
const install = require('./util').installPassport
```

```
install(app.passport, require('./passport'))
mount(['local'], app.passport, controller)
init('/api/v1', require('./api')(controller), router)
```

在 `install` 中添加验证逻辑，直接导入的是 `passport`。然后通过 `mount` 将 `local` 添加到路由上，跟以下代码类似，只不过抽象程度更高。

```
// 往控制器上面注册 passport 中间件
controller.passport = {}
controller.passport.local = app.passport.authenticate('local')
```

修改 `app/api.js`:

```
module.exports = ctx => ({
  post: {
    '/signup': ctx.user.signUp, // 注册
    '/signin': ctx.passport.local // 登录
  }
})
```

6. 添加 passport 逻辑

```
npm install debug
```

之前通过判断 `dev` 来输出和调试不是特别好用，换一个 `debug` 模块来试一试，其实还有更好的方法，就是将内置的 `logger` 逐一进行尝试，不断地演进架构。

添加 `app/passport/index.js`:

```
'use strict'
const passportDebug = require('debug')('app:passport')

module.exports = {
  async verify(ctx, user) {
    passportDebug('use ' + user.provider)
    return require('./' + user.provider)(ctx, user)
```





```

    },
    /**
     * * 在存储到 Session 或者 Cookie 前序列化
     * @param {BaseContext} ctx 上下文
     * @param {object} user 用户
     * @return {object} user
     */
    async serializeUser(ctx, user) {
      return user
    },
    /**
     * * 将序列化的数据还原
     * @param {BaseContext} ctx 上下文
     * @param {object} user 用户
     * @return {object} user
     */
    async deserializeUser(ctx, user) {
      return user
    }
  }
}

```

user.provider 就是 local、weibo、GitHub 等，这样就可以自动载入 local.js 或者 weibo.js。serializeUser 与 deserializeUser 写了暂时没用，因为没有用到 session。

想要开启上面的调试，只需要设置环境变量 `DEBUG=app:*` 就可以了，`app:*` 代表以 `app:` 开头的调试已开启。修改 `package.json` 的启动脚本，`cross-env` 是跨平台的环境变量设置工具，一些其他组件已经依赖它，所以不需要再安装。

```
"dev": "cross-env DEBUG=app* egg-bin dev"
```

7. 添加本地验证逻辑

创建 `app/passport/local.js`:

```

'use strict'
const R = require('ramda')
module.exports = async (ctx, { username, password }) => {
  const email = username
  await ctx.verify('user.signin', 'body')
  const user = await ctx.model.User.Auth(email, password)
}

```





```

ctx.assert(user, 400, '用户或密码错误')
const raw_user = R.omit(
  ['password', 'created_at', 'updated_at'],
  user.toJSON()
)
const token = await ctx.sign_token(raw_user, ctx.request.body.remember_me)
ctx.body = token
return token
}
if(!user){
  ctx.throw(400, '用户或密码错误')
}

ctx.assert(user, 400, '用户或密码错误')

```

两种方式其实是等价的，使用第二种方式可以简化代码。R.omit 可以删除第一个参数数组里的属性。通过 toJSON 方法把 ORM 实例转化为普通的 JS 对象。ctx.sign_token 用来生成 Token，即把用户信息记录到 Token 中。

给 ctx 添加 sign_token 方法，创建 app/extend/context.js:

```

'use strict'

module.exports = {
  async sign_token(user, remember_me) {
    return new Promise((resolve, reject) => {
      this.app.jwt.sign(
        user,
        this.app.config.jwt.secret,
        {
          expiresIn: remember_me ? '7d' : '1d'
        },
        (err, token) => {
          err && reject(err)
          resolve(token)
        }
      )
    })
  }
}

```





```
}  
}
```

`expiresIn` 为有效期，默认为一天，设置为记住密码则为 7 天。其实生成 Token 用的是 `jsonwebtoken` 库，而自动解析 Token 里的信息则用的是 `koa-jwt`。

8. 表单验证逻辑

创建 `app/schemas/user/signin.yml`：

```
# 用户登录  
email:  
  type: 'email'  
  required: true  
  message: '用户邮箱有误'  
password:  
  - min: 6  
    message: '密码长度不能小于 6 位'  
  - required: true  
    message: '请填写密码'  
remember_me:  
  type: boolean  
  message: '记住密码必须是一个布尔值'
```

9. 验证

修改 `home` 控制器，便于后面验证：

```
'use strict'  
  
class HomeController extends Controller {  
  async index() {  
    this.ctx.type = 'json'  
    this.ctx.body = this.ctx.state  
  }  
}  
  
module.exports = HomeController
```

打开 PostMan，使用上一节注册好的用户进行登录测试。



向 localhost:7001/api/v1/signin 提交 JSON 请求:

```
{
  "password": "8888888",
  "email": "ttttttt@qq.com"
}
```

结果如图 3-16 所示, 返回 Token。



图 3-16

控制台输出如下所示, app:passport 是由 debug 输出的, 而其他带方括号的部分是我们自定义的。

```
> nd
> miaoQ1.0.0 dev /Users/Yugo/workspace/nodejs_shizhan/chapter5/5-6/miao
> cross-env DEBUG=app* egg-bin dev

2018-02-22 13:07:34,804 INFO 5034 [master] node version v8.9.0
2018-02-22 13:07:34,947 INFO 5034 [master] egg version 2.3.0
2018-02-22 13:07:39,232 INFO 5035 [model] SELECT VERSION() as `version` (4ms)
2018-02-22 13:07:39,252 INFO 5035 [model] SELECT 1+1 AS result (6ms)
2018-02-22 13:07:39,262 INFO 5034 [master] agent_worker#1:5035 started (4298ms)
[ mount passport ] local
[ post ] -> /api/v1/signup
[ post ] -> /api/v1/signin
2018-02-22 13:07:43,633 INFO 5038 [model] SELECT VERSION() as `version` (19ms)
2018-02-22 13:07:43,674 INFO 5038 [model] SELECT 1+1 AS result (1ms)
2018-02-22 13:07:43,728 INFO 5034 [master] egg started on http://127.0.0.1:7001 (8777ms)
[ GET ] -> /api/v1/signin
app:passport use local +0ms
2018-02-22 13:09:14,288 INFO 5038 [model] SELECT `id`,`email`,`password`,`username`,`weibo`,`weixin`,`team_id`,`receive_remote`,`email_verified`,`avatar`,`created_at`,`updated_at` FROM `Users` AS `User` WHERE `User`.`email` = 'ttttttt@qq.com' LIMIT 1; (17ms)
{ id: 30,
  email: 'ttttttt@qq.com',
  username: 'bob',
  weibo: null,
  weixin: null,
  team_id: null,
  receive_remote: 0,
  email_verified: 0,
  avatar: null }
```

复制该 Token, 创建一个新的请求, 在 Authorization 中选择 Bearer Token, 填入 Token, 发送请求, 即可获取请求到的数据, 该数据为 sign_token 的 raw_user 信息, 即在 ctx.state 上的对象, 如图 3-17 所示。

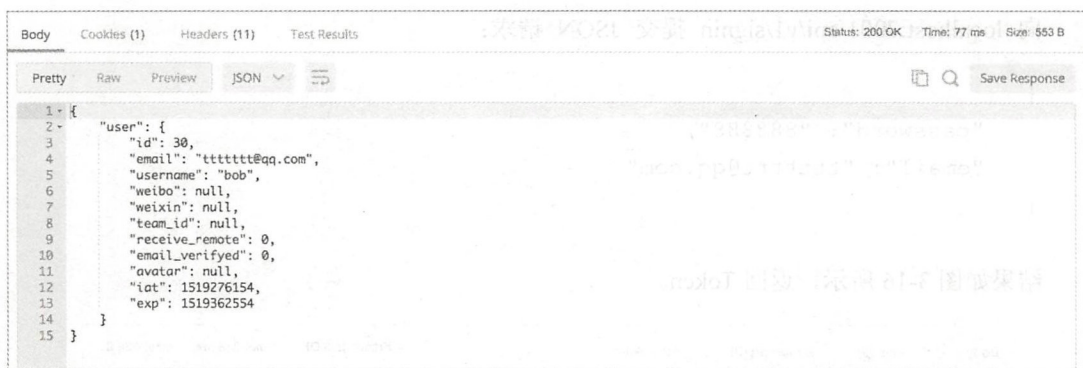


图 3-17

由于这个 Token 依赖登录，需要手动复制 Token，那么有没有办法让它不依赖登录？

在 Pre-request Script 中添加如下代码，通过 JS 请求登录接口，将 Token 设置为环境变量，然后在验证过程中通过 `{{ token }}` 进行调用。

```
pm.sendRequest({
  method: 'POST',
  url: 'http://127.0.0.1:7001/api/v1/signin',
  header: 'content-type:application/json',
  body: {
    mode: 'raw',
    raw: JSON.stringify({
      "password": "888888",
      "email": "ttttttt@qq.com"
    })
  }
}, function (err, response) {
  if (err) { console.log(err); }
  pm.environment.set("token", response.text());
});
```

结果如图 3-18 所示。

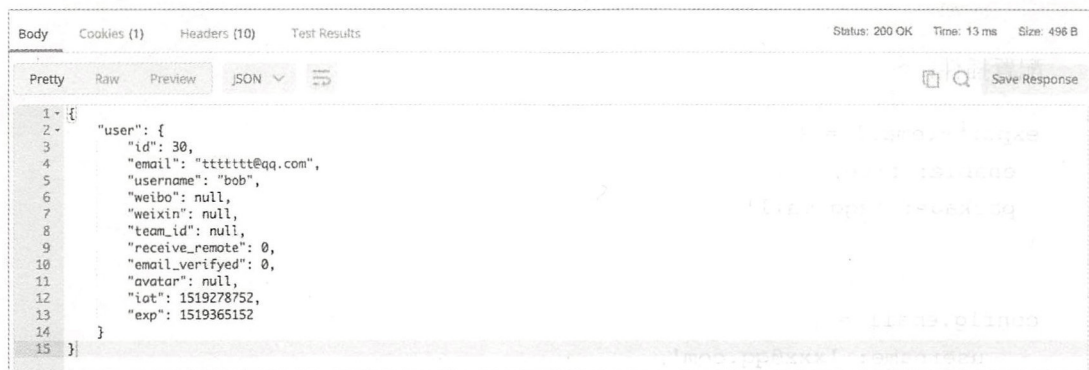


图 3-18

3.7 邮件与调试

发送邮件需要一个邮件服务器，通常来说搭建一个邮件服务器是完全没有必要的，个人可以直接使用个人邮箱，公司可以使用企业邮箱，这类服务也比较多，比如阿里云邮、Sendcloud 等，基本每个月都有几百条的免费额度。

3.7.1 理解发送邮件的原理

可以通过 HTTP 的角度去理解邮件服务器的一些协议。像 QQ 邮箱，我们在日常生活中会使用一些第三方客户端来接收邮件，在 QQ 邮箱的个人设置 -> 账户里。

- POP3: 用于从邮件服务器下载邮件；
- SMTP: 简单邮件传输协议，用于发送邮件；
- IMAP: 可理解为 POP3 升级版，不仅可以下载，还可以将邮件状态同步到邮件服务器中；
- Exchange: 微软发起的邮件联系人同步协议；
- CardDAV: 同步邮件联系人的服务。

POP3/SMTP 其实跟 HTTP 一样，就是一种协议，HTTP 尚且支持认证，SMTP 当然也支持，可以看到上图有一个生成授权码的链接，生成并保留它。我们只需要把它理解成 HTTP 的 API，这样我们获得授权之后，就可以到第三方客户端获取用户的邮件供用户阅读。

3.7.2 安装邮件插件

```
npm install egg-mail
```



配置插件：

```
exports.email = {
  enable: true,
  package: 'egg-mail'
}

config.email = {
  username: 'xxx@qq.com',
  password: 'xxxxxx',
  host: 'smtp.qq.com',
  sender: 'bob <xxx@qq.com>'
}
```

password 就是前面得到的授权码，sender 是指谁发送的。之后我们就可以使用 app.mail.sendEmail 来发送邮件。在 app.js 添加如下代码并测试：

```
const ret = await app.email.sendEmail('Title', 'Content', 'xxxxxx@qq.com')
info(ret)
```

以下是 ret 输出的结果，info 是我们全局添加的 debug 方法。

```
2018-02-23 13:14:46,401 INFO 12134 { code: 0,
  msg:
    { attachments: [],
      alternative: null,
      header:
        { 'message-id': '<1519362884942.0.12134@nodelover>',
          date: 'Fri, 23 Feb 2018 13:14:44 +0800',
          from: '=?UTF-8?Q?bob?=<1716857218@qq.com>',
          to: '1733996866@qq.com',
          subject: '=?UTF-8?Q?Title?=',
          content: 'text/plain; charset=utf-8',
          text: 'Content' } } }
```

3.7.3 环境与调试

1. 添加全局方法

修改 init.js:

```
function globalLogger(app) {
  if (DEV) {
```



```

    global.debug = app.logger.debug.bind(app.logger)
    global.info = app.logger.info.bind(app.logger)
  } else {
    global.info = () => {}
    global.debug = () => {}
  }
}

```

```

module.exports = {
  globalBaseInitial,
  globalLogger
}

```

修改 app.js:

```
const { globalBaseInitial, globalLogger } = require('./init')
```

```
globalBaseInitial(__dirname)
```

```

module.exports = async app => {
  globalLogger(app)
}

```

2. 修改之前的调试方式

对于 initRouterMap 方法，代码如下：

```

// if (DEV) {
//   console.log(
//     `${chalk.blue('[ ' + method + ' ]')}` -> `${chalk.red(prefix + url)}`
//   )
// }
// }
info('[%s] -> %s', method, chalk.red(prefix + url))

```

对于 mountPassportToController 方法，代码如下：

```

// if (DEV) {
//   console.log(`${chalk.blue('[ mount passport ]')}` ${chalk.red(value)})
// }

```



```
info('[passport] -> %s', chalk.red('controller.passport.' + value))
```

使用 VSCode 时请安装 Output Colorizer 扩展，打开 logs 下的日志文件会发现彩色的输出，对于多余的 “[35m” 其实是颜色代码，因为我们用 chalk 修改了颜色。

此时还在 debug 中，有些内容没有显示出来。这是日志级别的问题，Egg 的日志级别分为 NONE、DEBUG、INFO、WARN 和 ERROR。默认是 INFO 级别，只会显示 INFO、WARN 和 ERROR。

我们可以在本地开发环境中开启 debug，新建 config/config.local.js:

```
'use strict'

module.exports = () => {
  const config = {}
  config.logger = {
    consoleLevel: 'DEBUG',
    level: 'INFO'
  }
  return config
}
```

consoleLevel 为输出到控制台的级别，level 为输出到 logs 文件的级别。现在配置了本地开发环境选项，但是 Egg 并不知道当前是什么环境，我们可以在启动的时候通过 cross-env EGG_SERVER_ENV=local 来指定，也可以创建 config/env，内容为 local，这里我们采用创建文件的方式。之后打印 app.config.env 会看到 local。

3.7.4 全局调试

假如上面的调试还是无法满足要求，总是 log，每次 log 都要重启，太慢了，想要体验一秒看 5 个变量的“快感”怎么办？

首先通过 npm run debug 启动，打开所提示的 URL，类似以下链接。

```
chrome-devtools://devtools/bundled/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:9999/__ws_proxy__
```

可以看到如图 3-19 所示的页面，然后单击 sources 面板。

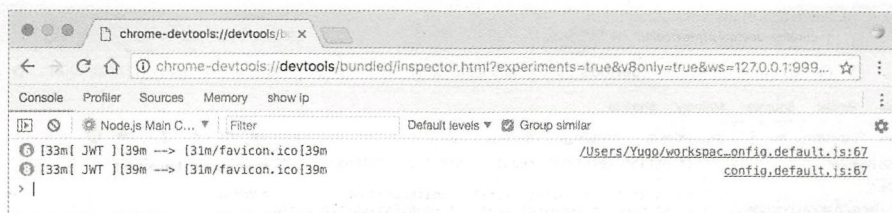


图 3-19

单击代码的行号可以打上断点，表示要停在这个地方，选中 `controller` 变量，单击鼠标右键选择 `add selected text to watches` 选项，如图 3-20 所示。

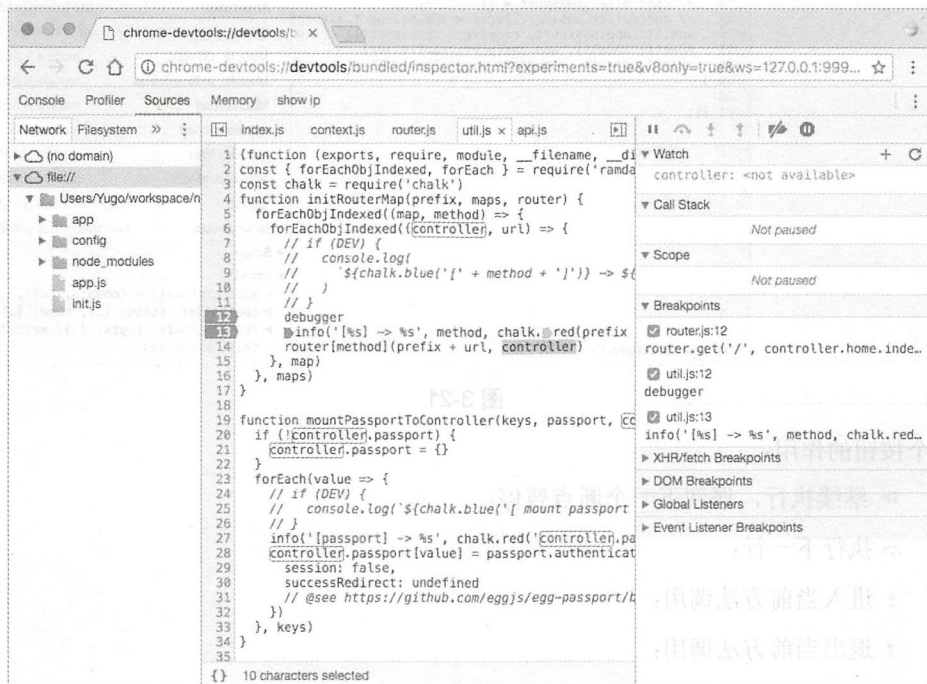


图 3-20

如图所示，会发现右边面板的 `watch` 栏里多了一个 `controller`。不过这个载入路由的逻辑，需要重启一下才能运行，所以我们随便修改一些代码让它重启，然后单击 `reconnect` 按钮即可。

可以看到如图 3-21 所示的调试信息。

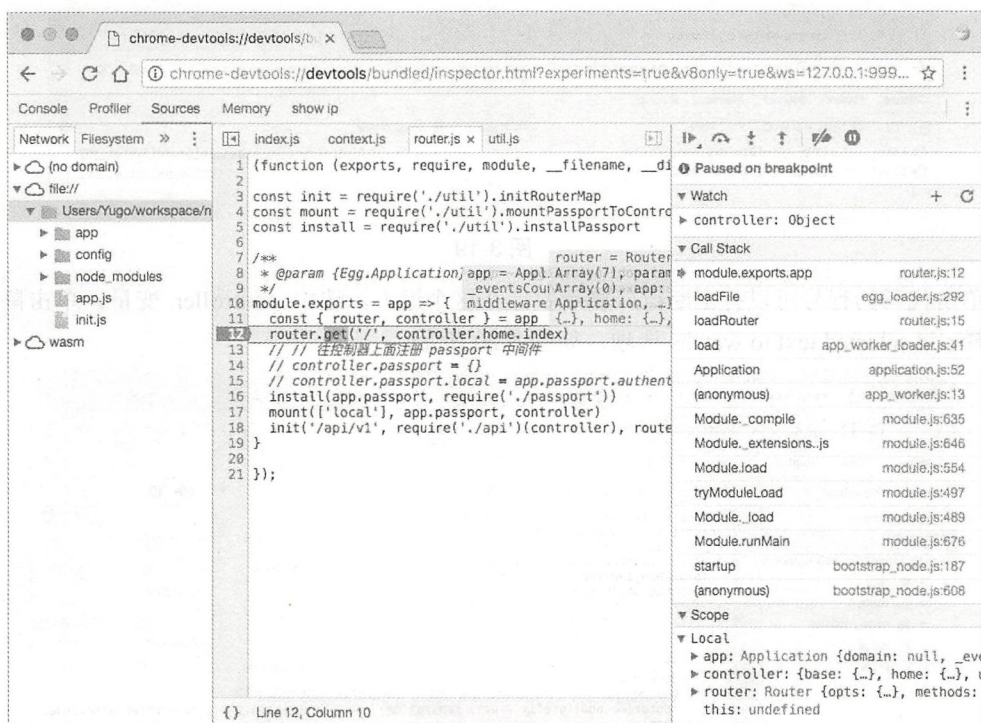


图 3-21

几个按钮的作用：

- ▶ 继续执行，遇到下一个断点暂停；
- ⏮ 执行下一行；
- ⏴ 进入当前方法调用；
- ⏵ 退出当前方法调用；
- 🚫 不捕获断点；
- 🛑 捕获异常。

想要捕获异常，可以打开捕获异常的选项，它在右边栏的上方，勾选 `Pause on caught exceptions` 选项，如图 3-22 所示。

访问首页，我们会捕获一个 401 的异常，如图 3-23 所示。

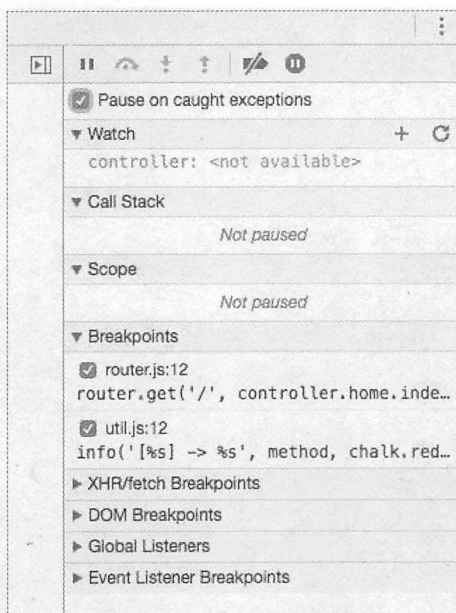


图 3-22

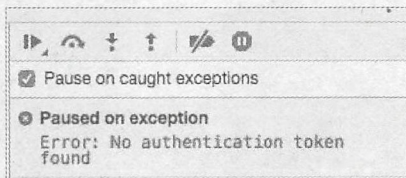


图 3-23

3.7.5 VSCode 全局调试

首先安装 `vscode-eggjs` 插件。

来到调试页面，单击配置旁边的小齿轮，选择 `Egg`，然后单击绿色小箭头运行。

错误 无法连接到运行中的进程，将在10000毫秒后超时 - (原因: 无法连接到目标: connect ECONNREFUSED 12... 关闭 打开 launch.json

通常来说会遇到一个错误，是因为超时。因为 VSCode 启动比较慢，我们在配置项 `JSON` 里添加一个超时的选项，编辑器会自动提示这个选项：

```
"timeout": 10000
```

然后打下一个断点，或者输入 `debugger` 来设置断点。使用 `debugger`，需要设置 `eslint` 的 `no-debugger` 为 `off`，如图 3-24 所示。

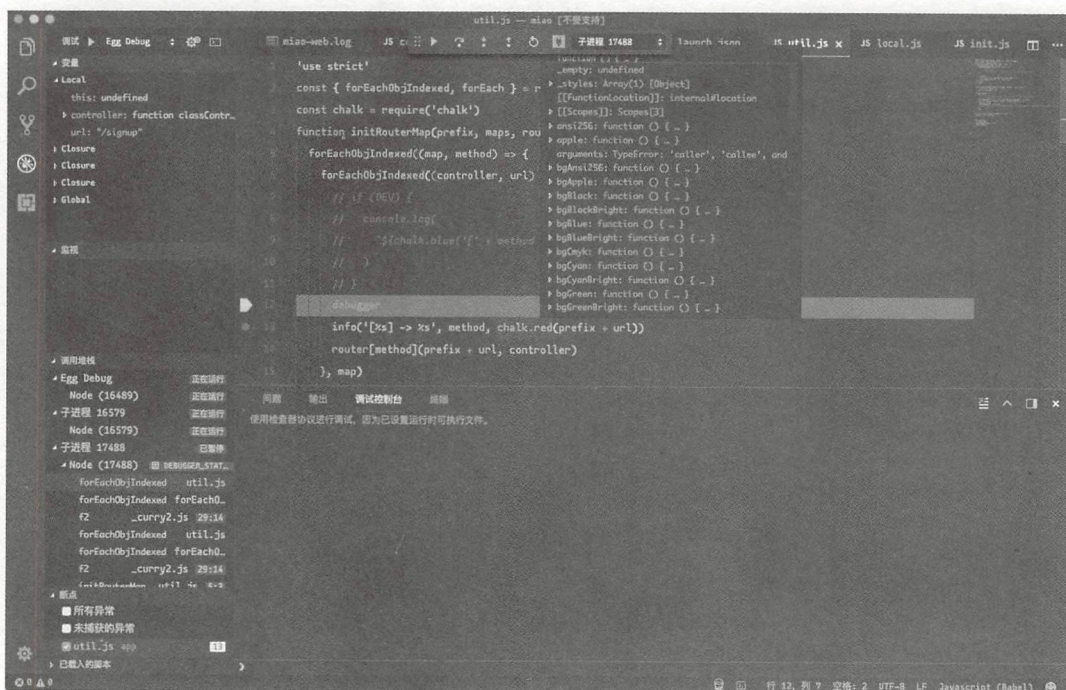


图 3-24

同样选中变量单击鼠标右键，可以添加到监视变量，如图 3-25 所示。

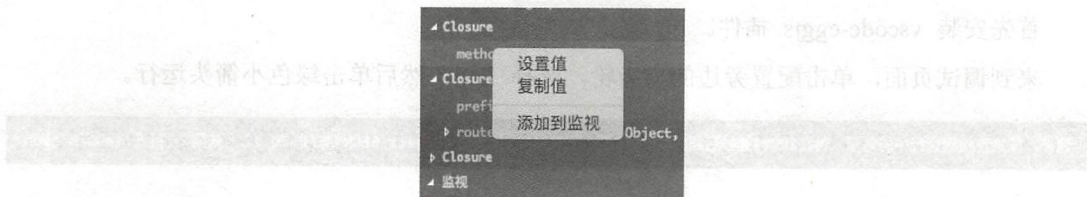


图 3-25

这里之所以有多个进程，是因为 Egg 要启动 Master、Agent 和 Worker。

3.7.6 发送验证邮件

1. 安装依赖

```
npm i egg-redis egg-view-nunjucks
```

Redis 也可以理解为数据库，不过它只能存储 key-value，就像一个大 Map 一样，即 `new Map()` == redis，同时它也是存在内存中的。

开启插件:

```
exports.redis = {
  enable: true,
  package: 'egg-redis'
}
exports.nunjucks = {
  enable: true,
  package: 'egg-view-nunjucks'
}
```

配置连接密码:

```
config.redis = {
  client: {
    port: 6379, // Redis port
    host: '127.0.0.1', // Redis host
    password: '',
    db: 0
  }
}
config.view = {
  defaultViewEngine: 'nunjucks',
  mapping: {
    '.njk': 'nunjucks'
  }
}
```

2. 为 VSCode 添加 nunjucks 代码片段

操作步骤如图 3-26 所示。



图 3-26

选择 `nunjucks.json`，将下面的内容复制进去。大致的功能就是，当你输入 `prefix` 中的内容时，它会出现 `body` 里面的代码。

```
{% %}: {
  "prefix": "cp",
  "body": [
    "{% $1 %}"
  ],
  "description": "控制块"
},
{% block %}: {
  "prefix": "block",
  "body": [
    "{% block $1 %}$2{% endblock %}"
  ],
  "description": "Block 块"
},
"extends": {
  "prefix": "et",
  "body": [
    "{% extends '$1' %}"
  ],
  "description": "继承块"
},
"{{ }}": {
  "prefix": "co",
  "body": [
    "{{ $1 }}"
  ],
  "description": "输出块"
}
```

3. 安装 Redis

通过下面的命令安装：

```
brew install redis
```

启动服务：

```
brew services start redis
```

3.7.7 添加逻辑

service

给 service/user.js 添加逻辑:

```
/**
 * 发送验证邮件
 * @param {(string)} title 标题
 * @param {(string| User)} E-mail 邮件地址
 * @param {Function} getTemplate 获取模板
 * @memberof User
 * @return {void}
 */
async sendVerifyEmail(title, email, getTemplate) {
  let user = null
  if (R.type(email) === 'Object') {
    user = email
    email = user.email
  } else {
    user = await this._User.findByEmail(email)
  }
  const token = this.ctx.random(4)
  await this.app.redis.set(
    'email:' + user.id,
    token,
    'EX',
    60 * 3 * 1000 // 3 分钟
  )
  const template = getTemplate(user, token)
  info('[send email template] -> ' + email + ' | ' + template)
  const mail_ret = await this.app.email.sendEmail(title, template, email)
  info(mail_ret)
  if (mail_ret.code === 0) {
    return true
  }
}
```



```

    }
    return false
  }
  /**
   * 验证邮件 Token
   * @param {number} user_id 用户 ID
   * @param {string} Token 验证码
   * @return {boolean} 通过否
   */
  async verifyToken(user_id, token) {
    const key = 'email:' + user_id
    const local_token = await this.app.redis.get(key)
    if (local_token === token) {
      await this.app.redis.del(key)
      const user = await this._User.findById(user_id)
      user.email_verified = 1
      await user.save()
      return true
    }
    return false
  }
}

```

- sendVerifyEmail 的 email 参数可以是 user 模型的实例，也可以是 email 邮箱，通过它们获取最终的 user，然后生成 4 个字符的随机串，存入 Redis，通过 getTemplate 参数取得发送邮件的模板并发送出去；
- verifyToken 则是从 Redis 里获取刚存入的 Token，然后进行比较，假如成功则返回 True。

1. 扩展

给 extend/context.js 添加随机字符串生成方法，这只是简单的随机字符串没必要用 UUID，因为存入 Redis 中的 key 还拼接了 user_id，所以这个 Token 随便是什么都可以，只要不是固定的值就行。

toString 方法的参数为进制，26 个字母加上 10 位数，所以是 36 位。因为前面三位是小数点，所以通过 slice 去掉小数点。

```

random(len) {

```

```
    return Math.random()
      .toString(36)
      .slice(3, len + 3)
  },
```

给 user 模型添加静态方法:

```
User.findByEmail = async function(email) {
  return await this.findOne({
    where: {
      email
    }
  })
}
```

修改 user 模型的定义, 保证邮件是唯一的, 记得把 migration 也修改一下。

```
email: {
  type: STRING(40),
  unique: true
},
```

2. 控制器

修改 controller/user.js:

```
signupTemplate(user, token) {
  return (
    '<a href="' +
    this.ctx.origin +
    '/email/verify?user_id=' +
    user.id +
    '&t=' +
    token +
    '">点我验证邮件</a>'
  )
}

/**
 * @description 注册
```

```
* @member User
*/
async signUp() {
  const { ctx } = this
  await ctx.verify('user.signup', 'body')
  const { user } = await ctx.service.user.signUp()
  const ok = await this.ctx.service.user.sendVerifyEmail(
    '激活邮箱',
    user,
    this.signUpTemplate.bind(this)
  )
  this.ok(ok)
}
/**
 * 激活验证
 * @return {void}
 */
async emailVerify() {
  await this.ctx.verify('user.emailVerify', 'query')
  const ok = await this.ctx.service.user.verifyToken(
    this.ctx.request.query.user_id,
    this.ctx.request.query.t
  )
  return this.ok(ok)
}
emailSendTemplate(user, token) {
  return (
    '<a href="' +
    this.ctx.origin +
    '/email/forget_password?user_id=' +
    user.id +
    '&t=' +
    token +
    '">点我验证邮件</a>'
  )
}
/**
 * 发送忘记邮件
```

```
* @return {void}
*/
async emailSend() {
  await this.ctx.verify('user.emailSend', 'body')
  const ok = await this.ctx.service.user.sendVerifyEmail(
    '找回密码',
    this.ctx.request.body.email,
    this.emailSendTemplate.bind(this)
  )
  return this.ok(ok)
}
/**
 * 忘记密码表单
 * @return {void}
 */
async forgetPasswordG() {
  return this.ctx.render('user/forget_password.njk')
}
/**
 * 忘记密码 POST 验证
 * @return {void}
 */
async forgetPasswordP() {
  const { ctx } = this
  const fields = Object.assign({}, ctx.request.query, ctx.request.body)
  await ctx.verify('user.forgetPassword', fields)
  const ok = await ctx.service.user.verifyToken(fields.user_id, fields.t)
  if (ok) {
    const user = await ctx.model.User.findById(fields.user_id)
    user.password = fields.password
    await user.save()
    return ctx.render('user/success.njk')
  }
  return ctx.render('user/failure.njk')
}
```

获取模板一定要用同步的方法，因为我们在 `service` 里没有用 `await`，暂时先这样，后续大家可以尝试改成从视图中渲染，通过 `renderView` 来获取。

3. 验证

schemas/user/emailSend.yml:

```
# 发送验证邮件
email:
  - type: 'email'
  - required: true
    message: '请提供用户邮箱'
```

schemas/user/emailVerify.yml:

```
# 验证邮件
user_id:
  required: true
  message: '请提供用户 ID'
t:
  required: true
  message: '请提供令牌'
# 忘记密码 POST 验证
user_id:
  required: true
  message: '请提供用户 ID'
t:
  required: true
  message: '请提供令牌'
password:
  - min: 6
    message: '密码长度不能小于 6 位'
  - required: true
    message: '请填写密码'
confirm_password:
  - required: true
    message: '请填写确认密码'
  - validator: !!js/function >
      function validator(ctx) {
        return async function (rule, value, callback, source, options) {
          if(ctx.request.body.password === value){
            return callback();
          }
        }
      }
```

```

    }
    callback([ { message: '密码与确认密码不匹配', field: 'confirm_password' } ])
  }
}

```

4. 添加视图

app/view/layout/app.njk:

```

<!DOCTYPE html>
<html lang="zh-CN">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>{% block title %} Miao Application {% endblock %}</title>
  <link rel="stylesheet" href="https://cdn.bootcss.com/bootstrap/4.0.0/
css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWv
TNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">
  <script src="https://cdn.bootcss.com/jquery/3.2.1/jquery.slim.min.js"
integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCKRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hX
pG5KkN" crossorigin="anonymous"></script>
  <script
src="https://cdn.bootcss.com/popper.js/1.12.9/umd/popper.min.js" integrity=
"sha384-ApNbgh9B+Y1QKt3Rn7W3mgPxmU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"
crossorigin="anonymous"></script>
  <script src="https://cdn.bootcss.com/bootstrap/4.0.0/js/bootstrap.min.js"
integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76
PVCmYl" crossorigin="anonymous"></script>
</head>
<body>
  {% include "./header.njk" %}
  {% block content %}{% endblock %}
  {% include "./footer.njk" %}
</body>
</html>

```

app/view/layout/footer.njk:

```

<div class="container-fluid">
  <div class="alert alert-light" role="alert">
    Footer
  </div>
</div>

```

app/view/layout/header.njk:

```

<nav class="navbar navbar-light bg-light">
  <a class="navbar-brand" href="#">
    
    Miao Application
  </a>
</nav>

```

app/view/user/failure.njk:

```

{% extends '../layout/app.njk' %}

{% block content %}
  <div class="container">
    <div class="alert alert-error" role="alert">无效令牌失败，请重新发送邮件</div>
  </div>

{% endblock %}

```

app/view/user/forget_password.njk:

```

{% extends "../layout/app.njk" %}

{% block content %}
  <div class="container">
    <form method="post">
      <input type='hidden' name="_csrf" value="{{ctx.csrf}}" />
      <div class="form-group">
        <label for="password"> 密码</label>
        <input type="password" class="form-control" name="password"

```

```

id="password" placeholder="密码">
</div>
<div class="form-group">
  <label for="confirm_password"> 密码</label>
  <input type="password" class="form-control" name="confirm_password"
id="confirm_password" placeholder="确认密码">
</div>
<button type="submit" class="btn btn-primary">提交</button>
</form>
</div>

```

```
{% endblock %}
```

app/view/user/success.njk:

```

{% extends '../layout/app.njk' %}

{% block content %}
  <div class="container">
    <div class="alert alert-primary" role="alert">修改密码成功</div>
  </div>

{% endblock %}

```

对于重置密码，使用单独的页面会方便一些，后续再考虑改成前端单页的形式，这里使用的是 Nunjuck，因为它支持继承模板，所以使用起来相对简单。

Egg 默认安装了 egg-view，我们又安装了 egg-view-nunjuck，这样就开启了 nunjuck 模板引擎支持，然后通过设置 mapping 将以 .njk 的结尾交给 nunjuck 模板引擎处理。

上面的 include 是指令，它表示引入文件，指令都是包裹在 {% %} 中的，而输出则使用 {{ }} 即可。基本指令都有一个匹配的 end 结束标识。而 extend 代表继承，block 代表可替换区块，如 success.njk 里定义的 content 区块会替换 app.njk 里的 content 区块，因为它是继承 app.njk 的，遇见更多的语法时笔者会再解释，想要了解更多，请到 <http://mozilla.github.io/nunjucks/> 阅读文档。

对于标签属性 integrity，其实它跟 Hash 一样，用于校验文件的完整性。在上面的 app.njk 中，笔者引入了 Bootstrap4 框架，并且是从 CDN 上引入的，读者可以到 <http://www.bootcdn.cn/> 查找自己想要的库进行引入。

使用 CDN 的好处就是，假如 A 网站使用了该 CDN 上的库，而 B 网站也使用该 CDN 上的同一个库。那么进入 A 网站之后，会缓存该库，之后进入 B 网站就不需要再次下载。

5. 路由

router.js:

```
router.get('/email/forget_password', controller.user.forgetPasswordG)
router.post('/email/forget_password', controller.user.forgetPasswordP)
router.get('/email/verify', controller.user.emailVerify)
```

api.js:

```
post: {
  '/email/send': ctl.user.emailSend
},
```

先暂时关闭所有的 JWT 来进行验证。

```
config.jwt = {
  secret: '123456',
  enable: true,
  ignore(ctx) {return true}
}
```

6. 兼容方案

context.js

重写 send 发送邮件的功能，Egg 插件的质量跟个人能力有关，egg-mail 目前来说还不支持发送 HTML，但是有人提交了 PR，还没有合并，所以笔者这里做一个临时的解决方案，当 egg-mail 支持之后，修改 send 方法调用 egg-mail 提供的 API 即可，目前先用这个自己写的功能。

这里返回的 code 是为了跟 egg-mail 保持一致的 API，假如不喜欢这种 Linux 错误码的风格，则可以直接用笔者写的这个方法，改一下 resolve，改成你所期望抛出的值即可。

```
const Email = require('emailjs/email')
const mailK = Symbol.for('MY_EMAIL')
async send(title, html, email, other) {
  const opts = this.app.config.email
```

```
if (!this[mailK]) {
  this[mailK] = Email.server.connect(
    Object.assign({}, opts, { user: opts.username, ssl: true })
  )
}

let attachment = [{ data: html, alternative: true }]
if (other) attachment = attachment.concat(other)
return new Promise((resolve, reject) => {
  this[mailK].send(
    {
      subject: title,
      from: opts.sender,
      to: email,
      attachment
    },
    (err, msg) => {
      if (err) reject({ code: -1, msg: 'faliure' })
      resolve({ code: 0, msg: 'success' })
    }
  )
})
},
```

3.7.8 验证

1. 注册用户

向 localhost:7001/api/v1/signup 发起 POST 请求，请自行准备好邀请码，然后填好数据，可以看到已经注册成功了，如图 3-27 所示。

也接收到了邮件，单击“验证”链接。

如果显示 success，则表示激活成功，当然也可以渲染一些好看的视图来替代这个丑陋的单词。

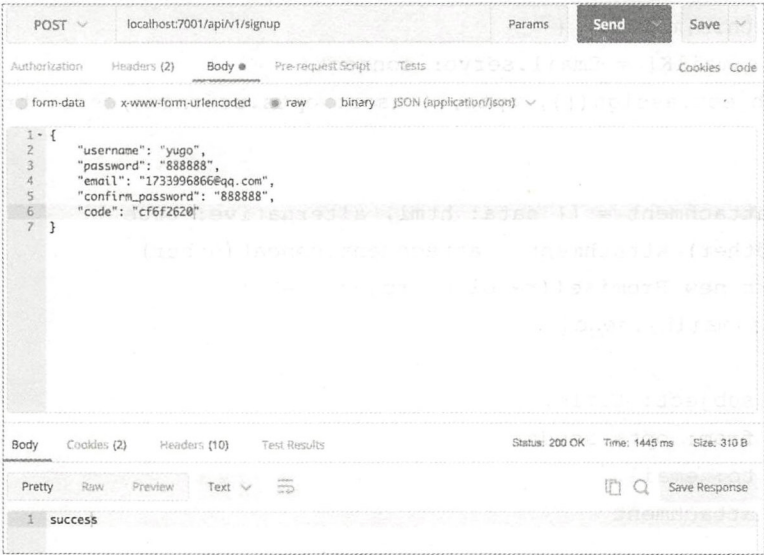


图 3-27

2. 找回密码功能

单击右上角的小眼睛，单击 Add 按钮或者 Edit 按钮，添加这两个变量，以后就可以批量更改了，如图 3-28 所示。在输入 URL 的地方通过 {{变量名}} 引用，跟 nunjuck 的使用方式一样。

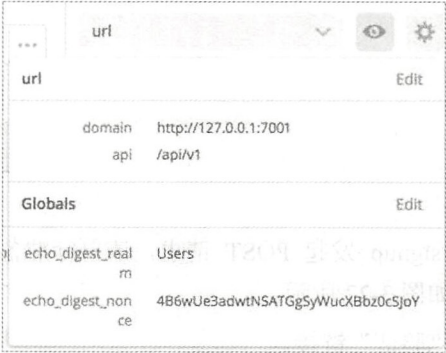


图 3-28

假如填错，则会跳到 failure 页面，所以验证功能不是特别完善，后续还需进一步完善。图 3-29 是成功的样例。

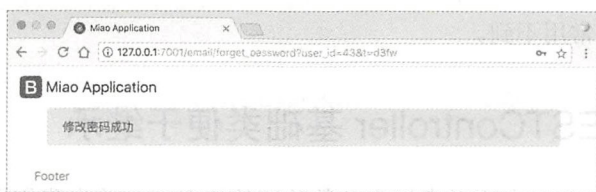


图 3-29

当我们通过浏览器历史回到上一级，再次提交会得到失效令牌的页面，如图 3-30 所示。



图 3-30

3.8 构建 RESTful API

3.8.1 什么是 RESTful API

以前很多企业里的 API 其实非常混乱，所以后来就有了一种 RESTful API 设计风格，将对一个资源的增删改查改成不同的 HTTP 请求方法，POST 表示创建，DELETE 表示删除，PUT 表示更新，GET 表示获取。

在 Egg 中通过 `router.resources` 可以快速构建 RESTful 风格的路由，表 3-1 是关于 posts 资源的 RESTful 风格的路由设计。

表 3-1 路由设计

Method	Path	Route Name	Controller.Action
GET	/posts	posts	app.controllers.posts.index
GET	/posts/new	new_post	app.controllers.posts.new
GET	/posts/:id	post	app.controllers.posts.show
GET	/posts/:id/edit	edit_post	app.controllers.posts.edit
POST	/posts	posts	app.controllers.posts.create
PUT	/posts/:id	post	app.controllers.posts.update
DELETE	/posts/:id	post	app.controllers.posts.destroy

其中 `new_post` 显示创建 post 的 HTML 视图，`edit_post` 是修改 post 的 HTML 视图，

对于构建 API 我们通常用不到。

3.8.2 创建 RESTController 基础类便于继承

为什么要使用基础类？在不考虑提交参数验证的情况下，首先我们思考对 Image 模型与 Post 模型的增删改查，都是调用同样的 create、destroy、update、findById、findAll 方法，唯一不一样的是模型，那么我们在构造器里通过名字获取这些模型不就行了吗？

1. 创建 app/controller/rest.js

```
'use strict'
const { defaultTo } = require('ramda')
const Base = require('./base')

class RESTController extends Base {
  constructor(ctx, modelName) {
    super(ctx)
    this.model = this.ctx.model[modelName]
  }
  /**
   * 留下的修改和删除的验证接口
   * @param {number} id model ID
   * @return {model} model instance
   */
  async getInstance(id) {
    const data = await this.model.findOne({ where: { id } })
    return data
  }
  /**
   * get list
   * @param {object} ctx Context
   * @param {object} where 所有条件但排除 order 和 include
   * @param {array} order 排序
   * @param {object} include 表连接
   */
  async index(ctx, where, order, include) {
    const { page, split } = ctx.query
    where = defaultTo({}, where)
```

Method	Path	Route Name	Controller/Action
GET	/posts	posts	get list
GET	/posts/new	new post	@param {object} ctx Context
GET	/posts/:id	post/:id	@param {object} where 所有条件但排除 order 和 include
GET	/posts/:id	edit post	@param {array} order 排序
POST	/posts	post	@param {object} include 表连接
PUT	/posts/:id	post/:id	
DELETE	/posts/:id	post/:id	

其中 new post 显示的是 post 的 HTML 结构



```
where.order = defaultTo([], order)
where.include = defaultTo([], include)
where.offset = defaultTo(0, parseInt(page) * parseInt(split))
where.limit = defaultTo(10, split)
info(where)
const data = await this.model.findAll(where)
ctx.body = data
}
/**
 * POST
 */
async create() {
  const { ctx } = this
  ctx.body = await this.model.create(ctx.request.body)
}
/**
 * Get one
 */
async show() {
  const { ctx } = this
  const data = await this.model.findOne({
    where: {
      id: ctx.params.id
    }
  })
  ctx.body = data
}
/**
 * Put
 */
async update() {
  const { id } = this.ctx.params
  const instance = await this.getInstance(id)
  Object.assign(instance, this.ctx.request.body)
  this.ctx.body = await instance.save()
}
/**
 * DELETE
```



```
    */
    async destroy() {
      const { id } = this.ctx.params
      const instance = await this.getInstance(id)
      this.ctx.body = await instance.destroy()
    }
  }

  module.exports = RESTController
```

这里使用的是 `require` 过来的 `Base`，是因为全局的 `Controller` 类型会丢失，这也是为什么 `Egg` 对 `TypeScript` 的支持不是特别友好，因为 `Egg` 使用的 `loader` 都是自动载入的，跟我们全局的 `Controller` 一样，但是好的一面是，官方比较重视 `TypeScript` 的支持，并为之进行了一些改造。

上面的 `constructor` 构造器需要我们传入 `modelName`，从 `this.ctx.model` 获得对应的模型。然后在其他方法里进行调用，之所以留下一个 `getInstance` 的方法，是因为在更新和删除的时候，可能有一些验证逻辑，直接重写这个方法即可。

2. app/controller/image.js

```
const REST = require('./rest')

class Image extends REST {
  constructor(ctx) {
    super(ctx, 'Image')
  }
}

module.exports = Image
```

这样就实现了一个自动 `RESTful` 控制器。对于一些特定的类型，比如 `Team`，它的逻辑会有些不一样，到时候重写方法即可。

3. 注册路由

`app/router.js`:

```
router.resources('images', '/images', controller.image)
```

第一个参数为路由的名字。



3.8.3 测试 Images RESTful API

1. POST 创建

创建方式如图 3-31 所示。

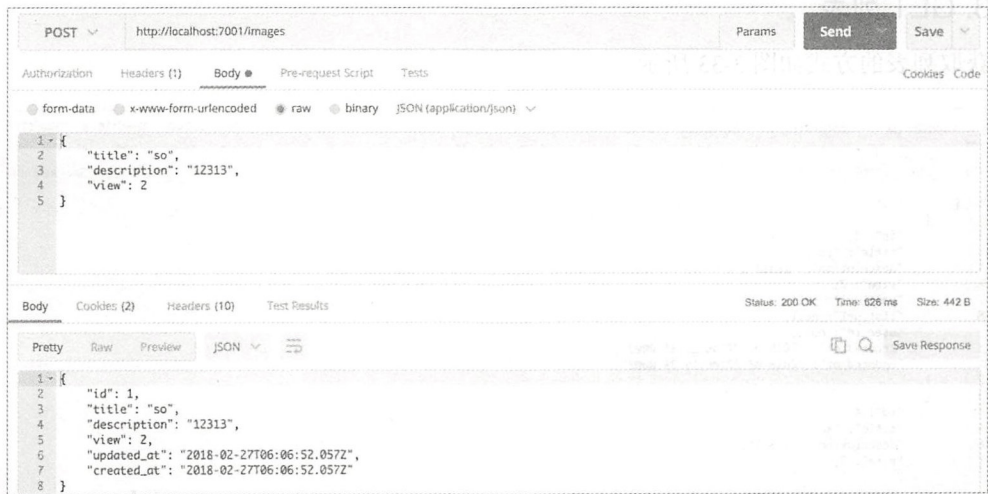


图 3-31

用 POST 向 `http://localhost:7001/images` 发送以下数据：

```
{
  "title": "so",
  "description": "12313",
  "view": 2
}
```

2. DELETE 删除

删除方式如图 3-32 所示。

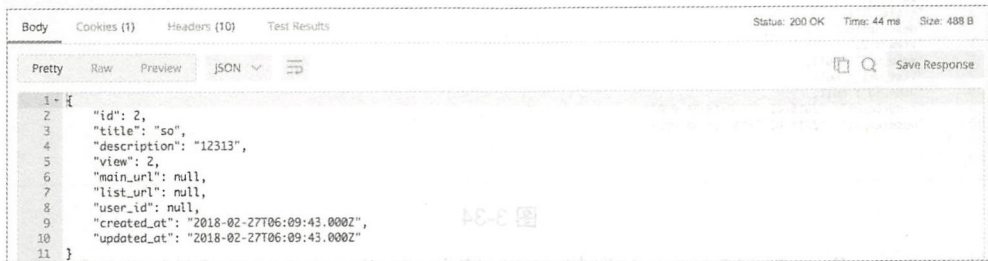


图 3-32



无论以何种方式发送请求都要设置请求头，设置 Content-Type 为 application/json，否则无法跳过 csrf token 的验证。

向 `http://localhost:7001/images/2` 发起 DELETE 请求，这里的 2 是 ID，一定要确保 ID 存在才可删除，应该是 Image 属于 user 才可以删除，不过目前我们还不考虑校验的问题。

3. GET 列表

获取列表的方式如图 3-33 所示。

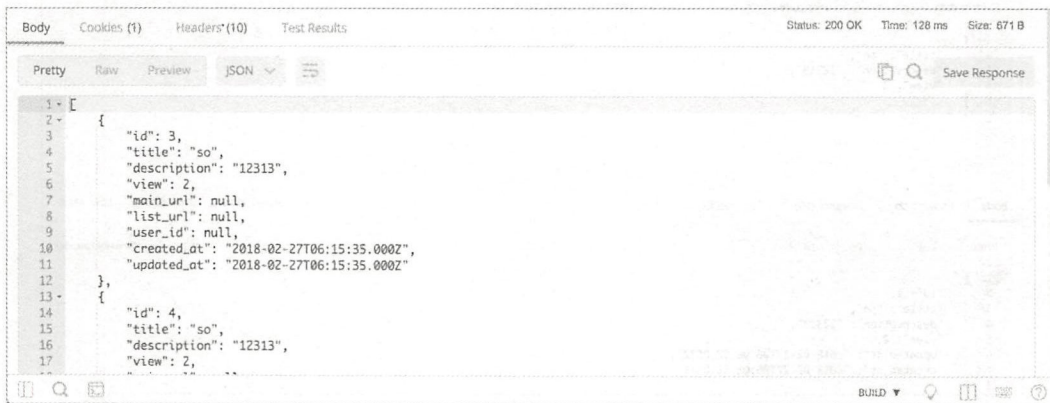


图 3-33

向 `http://localhost:7001/images` 发起 GET 请求，就可以看到数据。如果没有数据，则可以多发几条创建数据的 API，然后进行查询。

4. GET 单个数据

获取单个数据的方式如图 3-34 所示。

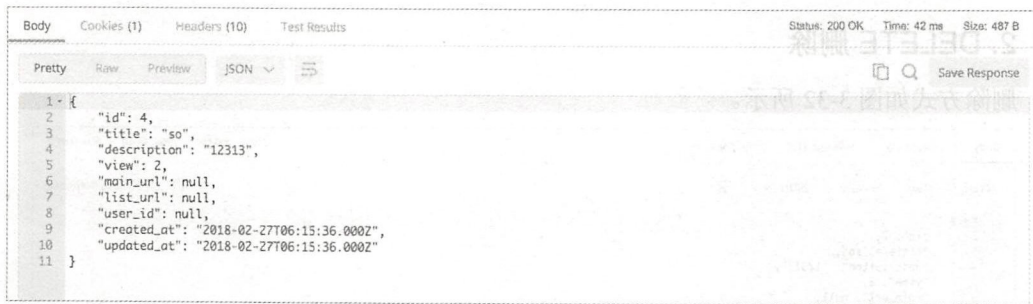


图 3-34

向 `http://localhost:7001/images/4` 发起 GET 请求，4 为 Image ID，同样可以看到数据。



5. PUT 修改

修改方式如图 3-35 所示。

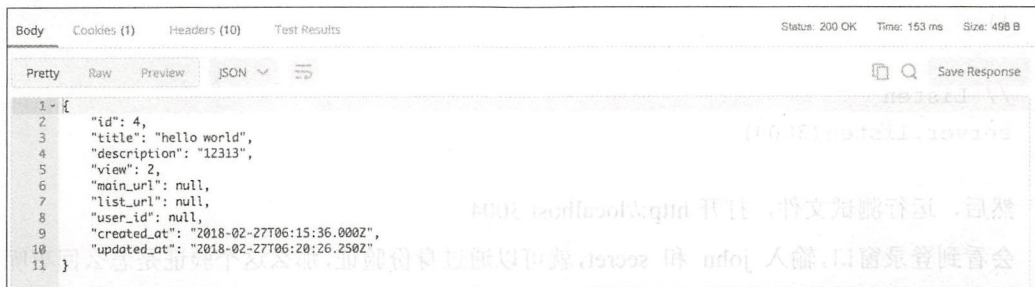


图 3-35

向 `http://localhost:7001/images/4` 发起 PUT 请求，将 `title` 改成 `hello world`。

3.8.4 构建后台的 REST 路由

1. 构建简单的 HTTP 验证

因为在后台可以对所有的模型都有修改权限，并且没有什么限制，所以我们要保护 API。这里用到了 `basic-auth`。

创建一个测试文件，代码如下：

```
var http = require('http')
var auth = require('basic-auth')

// Create server
var server = http.createServer(function(req, res) {
  var credentials = auth(req)
  console.log(credentials)
  if (
    !credentials ||
    credentials.name !== 'john' ||
    credentials.pass !== 'secret'
  ) {
    res.statusCode = 401
    res.setHeader('WWW-Authenticate', 'Basic realm="example"')
    res.end('Access denied')
```



```
    } else {  
      res.end('Access granted')  
    }  
  })  
}
```

```
// Listen  
server.listen(3004)
```

然后，运行测试文件，打开 <http://localhost:3004>。

会看到登录窗口，输入 `john` 和 `secret`，就可以通过身份验证，那么这个验证是怎么回事呢？当客户端第一次发送请求的时候，返回一个 `401` 状态码，并带上一个 `WWW-Authenticate` 的请求头，这样浏览器就会询问你的用户名和密码。在浏览器获得你的用户名和密码后，经过 `base64` 处理为你设置一个 `Authorization` 的请求头，然后 `base-auth` 会帮你把这个字段里的结果提取出来，如果正确就可以正常请求了，过程如图 3-36 所示。

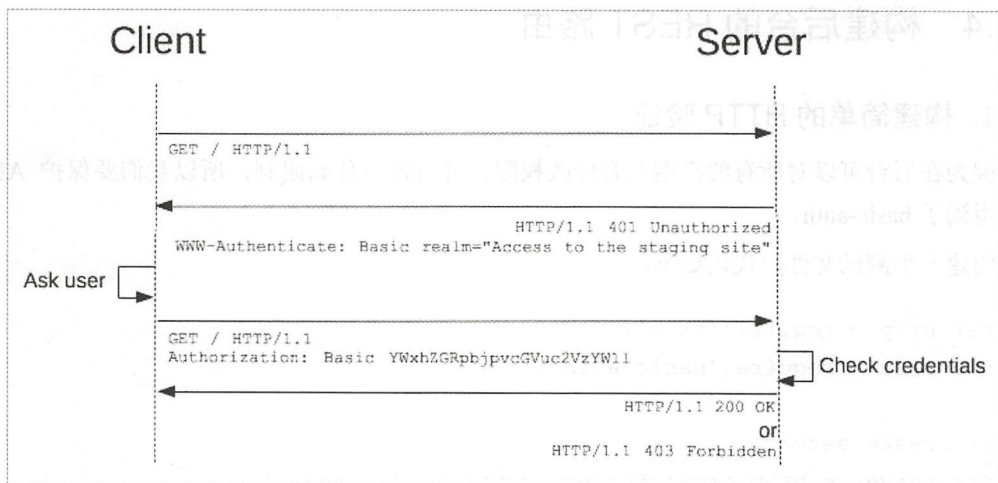


图 3-36

2. 构建 HTTP auth 的路由

安装依赖：

```
npm install koa-basic-auth  
const auth = require('koa-basic-auth')  
router.resources(  
  'admin',  
  '/api/v1/admin/:model',
```



```
auth({ name: 'dd', pass: '888888' })),  
controller.admin  
)
```

3.8.5 构建控制器

1. app/controller/all.js

```
'use strict'  
const Base = require('./base')  
const R = require('ramda')  
  
class AllModelController extends Base {  
  get model() {  
    const model = this.ctx.model.models[this.ctx.params.model]  
    info(model)  
    if (!model) {  
      return this.ctx.throw(400, '没有 ' + this.ctx.params.model + ' ! ')  
    }  
    return model  
  }  
  
  omit(filterArray = ['FM', 'Order', 'Team', 'User']) {  
    console.log(R.contains(this.ctx.params.model, filterArray))  
    if (R.contains(this.ctx.params.model, filterArray)) {  
      console.log(this.ctx.params.model)  
      return this.ctx.throw(403)  
    }  
  }  
  
  normarlize(data) {  
    const { ctx } = this  
    ctx.type = 'json'  
    ctx.body = {  
      data,  
      model: R.without(  
        ['FM', 'Order', 'Team', 'User'],  
        Object.keys(this.ctx.model.models)  
      )  
    }  
  }  
}
```




```
    }
  }
  /**
   * 留下的修改和删除的验证接口
   * @param {number} id model ID
   * @return {model} model instance
   */
  async getInstance(id) {
    const data = await this.model.findOne({ where: { id } })
    return data
  }
  /**
   * get list
   * @param {object} ctx Context
   * @param {object} where 所有条件但排除 order 和 include
   * @param {array} order 排序
   * @param {object} include 表连接
   */
  async index(ctx) {
    const where = ctx.request.body.where || {}
    this.omit(['Order', 'Team', 'User'])
    info(where)
    const data = await this.model.findAll(where)
    this.normarlize(data)
  }
  /**
   * POST
   */
  async create() {
    const { ctx } = this
    this.omit()
    const data = await this.model.create(ctx.request.body)
    this.normarlize(data)
  }
  /**
   * Get one
   */
  async show() {
```



```
const { ctx } = this
this.omit(['Order'])
const data = await this.model.findOne({
  where: {
    id: ctx.params.id
  }
})
this.normarlize(data)
}
/**
 * Put
 */
async update() {
  const { id } = this.ctx.params
  this.omit()
  const instance = await this.getInstance(id)
  Object.assign(instance, this.ctx.request.body)
  const data = await instance.save()
  this.normarlize(data)
}
/**
 * DELETE
 */
async destroy() {
  const { id } = this.ctx.params
  this.omit()
  const instance = await this.getInstance(id)
  const data = await instance.destroy()
  this.normarlize(data)
}
}

module.exports = AllModelController
```

all 控制器提供给前端使用，剔除了一些前端无法修改的表，all 的抽象程度比前面的 Image 更高，但是这也不安全，后续根据前端需求还需要再改。



2. app/controller/admin.js

```
'use strict'

const All = require('./all')

class Admin extends All {
  omit() {}

  normarlize(data) {
    const { ctx } = this
    ctx.type = 'json'
    ctx.body = {
      data,
      model: Object.keys(this.ctx.model.models)
    }
  }
}

module.exports = Admin
```

我们希望后端没那么多限制，可以通过 `http-auth` 来保证安全，所以这一层的校验可以重写。

3.8.6 测试后台路由

我们只简单地测试了 Post，选择 Basic Auth，输入之前设置的密码即可，如图 3-37 所示。

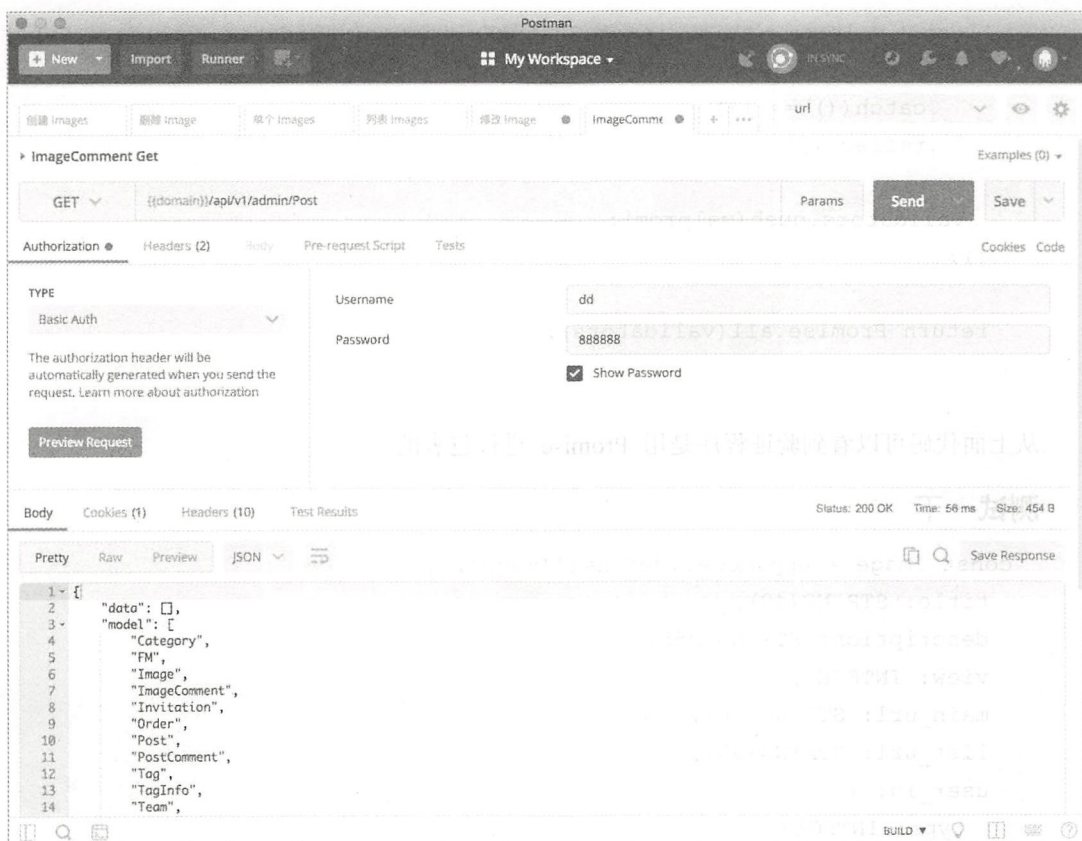


图 3-37

3.8.7 关于验证

其实在 ORM 的 Sequelize 中就提供了一些字段验证的选项，笔者进入源码搜寻一番后发现它是支持异步的。

```
_customValidators() {  
  const validators = [];  
  _.each(this.modelInstance._modelOptions.validate, (validator, validatorType)  
=> {  
    if (this.options.skip.indexOf(validatorType) >= 0) {  
      return;  
    }  
  })  
}
```



```
const valprom = this._invokeCustomValidator(validator, validatorType)
// errors are handled in settling, stub this
.catch(() => {})
.reflect();

validators.push(valprom);
});

return Promise.all(validators);
}
```

从上面代码可以看到验证程序是用 Promise 进行包裹的。

测试一下

```
const Image = app.model.define('Image', {
  title: STRING(40),
  description: STRING(255),
  view: INTEGER,
  main_url: STRING(20),
  list_url: STRING(90),
  user_id: {
    type: INTEGER,
    validate: {
      async isSome() {
        const name = await Promise.resolve('123')
        console.log(name)
        return true
      }
    }
  }
})
```

```
123
2018-02-27 16:16:18,640 INFO 12381 [model] INSERT INTO "Image"
(1, '2018-02-27 08:16:18', '2018-02-27 08:16:18'); (43ms)
```

可以发现在执行 SQL 之前，打印出了 123，假如我们的规则依赖上下文怎么办？这里提供几个思路，用一个 service，在其中有规律地实现对应的方法，比如 Image 的 PUT，那么

名字就为 `putImage`，只要有一定规律就行，这个规律就是我们的约定，然后在控制器里调用这个方法。或者把这个方法放在 `ORM` 的实例方法上。

3.9 安全地开放 API

如何对 API 进行有效的保护是我们开放 API 面临的第一道难关，通过 `OAuth 2.0` 就可以解决这个问题，那么 `OAuth 2.0` 是什么？它能解决什么问题？

1. 了解 OAuth 2.0

`2.0` 的意思就是第二代的版本，`OAuth` 是一种认证的标准，它有很多种类型，比如授权码模式、密码模式、客户端模式、简化模式等。这里只对授权码模式进行详细说明，因为它是最安全的模式，也是最复杂的模式。在后端实现了 `OAuth` 服务，就可以让其他网站使用我们的账号登录，但是用户又不需要把账户交给其他网站。我们平常见到的 `QQ` 登录就是典型应用案例。本质上来说就是两台服务器之间通过 `HTTP` 相互交换数据而已。

2. 授权码登录的流程

实现了 `OAuth` 之后，可以给很多的网站提供我们的账号登录权限，为了避免网站之间发生冲突，我们要用一个表来区分它们，用 `clint_id` 和 `clientSecret` 两个字段来进行区分。

首先将想要使用我们账户登录的网站称为 `A` 网站，我们的网站称为喵览。当用户单击 `A` 网站的喵览账户登录时，`A` 网站重定向到喵览网站的 `OAuth` 认证页面，并带上 `A` 网站在喵览上申请好的 `client_id` 和 `clientSecret`，以及回调的 `URL`。这个回调的 `URL` 就像一个邮件地址，当用户确定授权之后，喵览会把授权码发送到这个 `URL` 地址。为了安全，在喵览申请 `client_id` 和 `clientSecret` 的页面时，还应该设置安全回调 `URL`，每次都校验 `A` 网站提交过来的回调 `URL`，因为客户有可能会修改这个回调地址而取得授权码。

接下来喵览的用户正常登录，登录之后出现一个选项框，表示是否对 `A` 网站进行授权，比如说访问用户个人资料等，用户表示给予授权之后，喵览给 `A` 网站的回调地址发送授权码。当 `A` 网站接收到授权码之后，可以继续向喵览的授权接口发送请求，得到资源访问码与用户的个人数据，那么通过像之前小节中的 `JWT` 一样设置 `Authorization` 头，就可以访问受保护的资源与 `API` 了。

3. 实现数据结构表

生成 `Client Model`:

```
npx sequelize model:create --name Client --attributes client_id:integer,
user_id:integer,client_secret:string,redirect_uris:string,grants:string
```

修改 Client Model:

```
module.exports = app => {
  const DataTypes = app.Sequelize
  const Client = app.model.define(
    'Client',
    {
      client_id: {
        type: DataTypes.INTEGER,
        unique: true
      },
      user_id: DataTypes.INTEGER,
      client_secret: DataTypes.STRING,
      redirect_uris: DataTypes.STRING,
      grants: DataTypes.STRING
    },
    {
      underscored: false
    }
  )
  Client.prototype.associate = function() {
    this.belongsTo(app.model.User)
    app.model.User.hasOne(this)
  }
  Client.findByClientId = id =>
    Client.findOne({
      where: {
        client_id: id
      }
    })

  Client.Auth = (client_id, client_secret) => {
    const where = client_secret
      ? { where: { client_id, client_secret } }
      : { where: { client_id } }
    return Client.findOne(where)
  }
}
```

```

    return Client
  }

```

将其修改成 Egg 标准的 model 格式。grants 是授权模式,比如授权码模式,redirect_uri 是安全地回调 URL。

由于生成的时间戳字段为 createAt, 而 Egg sequelize 的配置是带下划线的, 所以查询会出错, 在模型定义的级别将 underscored 设置为不带下划线, 或者修改一下 migration。

生成 Authorization Model:

```

npx sequelize model:create --name Authorization --attributes code:string,
expires_at:date,redirect_uri:string,scope:string,client_id:string,user_id:integer

```

修改 Authorization Model:

```

module.exports = function(app) {
  const DataTypes = app.Sequelize
  const Authorization = app.model.define(
    'Authorization',
    {
      code: DataTypes.STRING,
      expires_at: DataTypes.DATE,
      redirect_uri: DataTypes.STRING,
      scope: DataTypes.STRING,
      client_id: DataTypes.STRING,
      user_id: DataTypes.INTEGER
    },
    {
      underscored: false
    }
  )
  Object.assign(Authorization, {
    async findByCode(code) {
      return this.findOne({
        where: {
          code

```



```

    }
  })
}
})
return Authorization
}

```

Authorization 用来存储授权码，expires_at 是有效期，scope 是授权范围，现在我们就进行关系的关联了，生成的方法实在难以记忆，还不如手动控制。

创建 Access:

```

npx sequelize model:create --name Access --attributes token:string,
token_expires_at:date,scope:string,client_id:integer,user_id:integer

```

修改 Access 模型:

```

module.exports = ({ model: sequelize, Sequelize: DataTypes }) => {
  const Access = sequelize.define(
    'Access',
    {
      token: {
        type: DataTypes.STRING,
        unique: true
      },
      token_expires_at: DataTypes.DATE,
      scope: DataTypes.STRING,
      client_id: DataTypes.INTEGER,
      user_id: DataTypes.INTEGER
    },
    {
      underscored: false
    }
  )
  return Access
}

```

Access 模型用来存储访问码。

创建 Refresh:

```
npx sequelize model:create --name Refresh --attributes token:string,
token_expires_at:date,scope:string,client_id:integer,user_id:integer
```

修改 Refresh 模型:

```
module.exports = ({ model: sequelize, DataTypes }) => {
  const Refresh = sequelize.define(
    'Refresh',
    {
      token: {
        type: DataTypes.STRING,
        unique: true
      },
      token_expires_at: DataTypes.DATE,
      scope: DataTypes.STRING,
      client_id: DataTypes.INTEGER,
      user_id: DataTypes.INTEGER
    },
    {
      underscored: false
    }
  )
  Refresh.findByToken = token => {
    return Refresh.findOne({
      where: {
        token
      }
    })
  }
  return Refresh
}
```

Refresh 模型用来存储刷新 Token，为了支持刷新 access_token，通常将 access_token 有效期设置较短，当过期后，可以通过 refresh_token 来重新获取 access_token。

创建表

查看状态，笔者尝试用 coffeescript 来写 migration，发现也是可以成功的，感兴趣的读者

可以阅读源码，然后在运行的时候加上 `coffee` 标志就可以。由于在这里 `coffeescript` 依赖 `js2coffee` 和 `coffee-script`，所以要通过 `npm` 进行安装，然后运行以下命令。

```
npx sequelize db:migrate:status -coffee
```

查看当前的状态，如下：

```
up    20180205101006-create-user.js
up    20180206052614-create-invitation.js
up    20180206053034-create-image.js
up    20180206053041-create-image-comment.js
up    20180206053046-create-post.js
up    20180206053051-create-post-comment.js
up    20180206053056-create-category.js
up    20180206053100-create-tag.js
up    20180206053106-create-tag-info.js
up    20180206053111-create-fm.js
up    20180206053117-create-team.js
up    20180206053124-create-team-status.js
up    20180206053136-create-order.js
down  20180301041230-create-client.js
down  20180301042017-create-authorization.js
down  20180301054400-create-access.coffee
down  20180301055252-create-refresh.coffee
```

```
npx sequelize db:migrate -coffee
```

进行表填充，运行结果如下：

```
Sequelize [Node: 8.9.0, CLI: 2.8.0, ORM: 4.33.4]

Loaded configuration file "config/config.json".
Using environment "development".
Thu, 01 Mar 2018 06:11:00 GMT sequelize deprecated String based operators
Please use Symbol based operators for better security, read more
manual/tutorial/querying.html#operators at node_modules/sequelize
== 20180301041230-create-client: migrating =====
== 20180301041230-create-client: migrated (0.374s)
== 20180301042017-create-authorization: migrating =====
== 20180301042017-create-authorization: migrated (0.433s)
== 20180301054400-create-access: migrating =====
You have to add "coffee-script" to your package.json.
```

```
Sequelize [Node: 8.9.0, CLI: 2.8.0, ORM: 4.33.4]

Loaded configuration file "config/config.json".
Using environment "development".
== 20180301054400-create-access: migrating =====
== 20180301054400-create-access: migrated (0.293s)
== 20180301055252-create-refresh: migrating =====
== 20180301055252-create-refresh: migrated (0.379s)
```

这里会有一个 sequelize operator 的警告，配置 config/config.json 即可忽略。

```
"development": {
  "username": "root",
  "password": null,
  "database": "miao",
  "host": "127.0.0.1",
  "dialect": "mysql",
  "operatorsAliases": false
},
```

4. 安装插件

```
npm install egg-outh2-server
```

egg-oauth2-server 是对 node-oauth2-server 的一个封装，node-oauth2-server 中定义好了如何实现上面所述的授权码模式的授权，只需要我们实现对应的接口即可。

开启插件：

```
exports.oAuth2Server = {
  enable: true,
  package: 'egg-oauth2-server'
}
```

配置插件：

```
config.oAuth2Server = {
  debug: true,
  grants: ['authorization_code', 'refresh_token']
}
```

只开启授权码模式和刷新 Token 模式。

创建实现接口

extend/oauth.js:

```
'use strict'
```




```
module.exports = app => {
  return class {
    constructor(ctx) {
      this.ctx = ctx
    }
  }
}
```

3.10 实现 OAuth 接口

3.10.1 实现授权码官方文档所要求的接口

接口的文档在 <https://oauth2-server.readthedocs.io/en/latest/model/overview.html#authorization-code-grant>, 以及 <https://github.com/Azard/egg-oauth2-server> 中可以找到。

要实现授权码授权就必须实现上面的这些方法。generator 开头的方法都是可选的，因为内置了生成随机字符串的方法，所以就没必要重写，对于特殊需求我们才重写这些方法，比如对用户的信息通过 jsonwebtoken 进行签名。

图 3-38 描述了生命周期，authorize 是发放授权码的 API 接口，token 是发放访问令牌的 API，或者说通过授权码交换访问令牌的接口，而 authenticate 是用于保护路由的 API。

```
authorization_code mode app.oauth.authorize() lifecycle
  getClient --> getUser --> saveAuthorizationCode

authorization_code mode app.oauth.token() lifecycle
  getClient --> getAuthorizationCode --> saveToken --> revokeAuthorizationCode

authorization_code mode app.oauth.authenticate() lifecycle
  Only getAccessToken
```

图 3-38

方法漏实现了也没关系，会有错误提示哪个所需的方法没有实现。

1. 实现 getAuthorizationCode

查询通过 saveAuthorizationCode 存储过的授权码并返回。可见 saveAuthorizationCode 稍后我们也要实现。



官方实现的版本

```
function getAuthorizationCode(authorizationCode) {
  // imaginary DB queries
  db.queryAuthorizationCode({authorization_code: authorizationCode})
    .then(function(code) {
      return Promise.all([
        code,
        db.queryClient({id: code.client_id}),
        db.queryUser({id: code.user_id})
      ]);
    })
    .spread(function(code, client, user) {
      return {
        code: code.authorization_code,
        expiresAt: code.expires_at,
        redirectUri: code.redirect_uri,
        scope: code.scope,
        client: client, // with 'id' property
        user: user
      };
    });
}
```

我们来实现一个比 `async` 更优雅的 `getAuthorizationCode`，官方给的例子返回的字段名叫什么，我们就原样返回什么，要不然改乱了，就找不到“北”了。

我们实现的版本

给 `Authorization` 添加方法：

```
Object.assign(Authorization, {
  async findByCode(code) {
    return this.findOne({
      where: {
        code
      }
    })
  }
})
```



给 Client 添加方法：

```
Client.findByClnetId = id =>
  Client.findOne({
    where: {
      client_id: id
    }
  })
```

两种风格都差不多。

```
'use strict'
```

```
module.exports = app => {
  const { User, Authorization, Client, Refresh, Access } = app.model
  console.log(app.model.models)
  return class {
    constructor(ctx) {
      this.ctx = ctx
      console.log(ctx.model)
    }

    async getAuthorizationCode(authorizationCode) {
      const auth = await Authorization.findByCode(authorizationCode)
      const [client, user] = await Promise.all([
        Client.findByClnetId(auth.client_id),
        User.findById(auth.user_id)
      ])
      return {
        code: auth.code,
        expiresAt: auth.expires_at,
        redirectUri: auth.redirect_uri,
        scope: auth.scope,
        client,
        user
      }
    }
  }
}
```



2. 实现 getClient

获取对应的 Client，便于后面校验 scope。

官方实现的版本

```
function getClient(clientId, clientSecret) {  
  // imaginary DB query  
  let params = {client_id: clientId};  
  if (clientSecret) {  
    params.client_secret = clientSecret;  
  }  
  db.queryClient(params)  
    .then(function(client) {  
      return {  
        id: client.id,  
        redirectUris: client.redirect_uris,  
        grants: client.grants  
      };  
    })  
}
```

对照着官方的代码实现比较容易，而看返回值的类型再实现则不会那么容易。

我们实现的版本

给 Client 模型添加方法：

```
Client.Auth = (client_id, client_secret) => {  
  const where = client_secret  
    ? { where: { client_id, client_secret } }  
    : { where: { client_id } }  
  return Client.findOne(where)  
}
```

添加方法：

```
async getClient(client_id, client_secret) {  
  const client = await Client.Auth(client_id, client_secret)  
  if (!client) return false  
  return {
```




```

    id: client.client_id,
    redirectUri: client.redirect_uris.split(','),
    grants: client.grants.split(',')
  }
}

```

`redirect_uris` 和 `grants` 都是通过 `,` 分割字符串，通过 `split` 将它们变成数组。

3. 实现 `saveToken`

保存 Token 令牌，包括访问 Token 令牌和刷新 Token 令牌。

官方实现的版本

```

function saveToken(token, client, user) {
  // imaginary DB queries
  let fns = [
    db.saveAccessToken({
      access_token: token.accessToken,
      expires_at: token.accessTokenExpiresAt,
      scope: token.scope,
      client_id: client.id,
      user_id: user.id
    }),
    db.saveRefreshToken({
      refresh_token: token.refreshToken,
      expires_at: token.refreshTokenExpiresAt,
      scope: token.scope,
      client_id: client.id,
      user_id: user.id
    })
  ];
  return Promise.all(fns);
  .spread(function(accessToken, refreshToken) {
    return {
      accessToken: accessToken.accessToken,
      accessTokenExpiresAt: accessToken.expires_at,
      refreshToken: refreshToken.refresh_token,
      refreshTokenExpiresAt: refreshToken.expires_at,

```



```

    scope: accessToken.scope,
    client: {id: accessToken.client_id},
    user: {id: accessToken.user_id}
  });
});
}

```

我们实现的版本

```

async saveToken(token, client, user) {
  const access = await Access.create({
    token: token.accessToken,
    token_expires_at: token.accessTokenExpiresAt,
    scope: token.scope,
    client_id: client.id,
    user_id: user.id
  })
  const refresh = await Refresh.create({
    token: token.refreshToken,
    token_expires_at: token.refreshTokenExpiresAt,
    scope: token.scope,
    client_id: client.id,
    user_id: user.id
  })
  return {
    accessToken: access.token,
    accessTokenExpiresAt: access.token_expires_at,
    refreshToken: refresh.token,
    refreshTokenExpiresAt: refresh.token_expires_at,
    scope: access.scope,
    client: { id: access.client_id },
    user: { id: access.user_id }
  }
}

```

4. 实现 saveAuthorizationCode

保存授权码。



官方实现的版本

```
function saveAuthorizationCode(code, client, user) {  
  // imaginary DB queries  
  let authCode = {  
    authorization_code: code.authorizationCode,  
    expires_at: code.expiresAt,  
    redirect_uri: code.redirectUri,  
    scope: code.scope,  
    client_id: client.id,  
    user_id: user.id  
  };  
  return db.saveAuthorizationCode(authCode)  
    .then(function(authorizationCode) {  
      return {  
        authorizationCode: authorizationCode.authorization_code,  
        expiresAt: authorizationCode.expires_at,  
        redirectUri: authorizationCode.redirect_uri,  
        scope: authorizationCode.scope,  
        client: {id: authorizationCode.client_id},  
        user: {id: authorizationCode.user_id}  
      };  
    });  
}
```

我们实现的版本

```
async saveAuthorizationCode(code, client, user) {  
  const auth = await Authorization.create({  
    code: code.authorizationCode,  
    expires_at: code.expiresAt,  
    redirect_uri: code.redirectUri,  
    scope: code.scope,  
    client_id: client.id,  
    user_id: user.id  
  })  
  
  return {  
    authorizationCode: auth.code,
```



```

    expiresAt: auth.expires_at,
    redirectUri: auth.redirect_uri,
    scope: auth.scope,
    client: { id: auth.client_id },
    user: { id: auth.user_id }
  }
}

```

5. 实现 revokeAuthorizationCode

当授权之后，应该吊销授权码。

官方实现的版本

```

function revokeAuthorizationCode(code) {
  // imaginary DB queries
  return db.deleteAuthorizationCode({authorization_code: code.authorizationCode})
    .then(function(authorizationCode) {
      return !!authorizationCode;
    });
}

```

我们实现的版本

```

async revokeAuthorizationCode(code) {
  const auth = await Authorization.findByCode(code.code)
  if (!auth) return true
  return await auth.destroy()
}

```

6. 实现 validateScope

校验 scope 是否符合规范。

官方实现的版本

```

const VALID_SCOPES = ['read', 'write'];

function validateScope(user, client, scope) {
  return scope
    .split(' ')
    .filter(s => VALID_SCOPES.indexOf(s) >= 0)
}

```




```

    .join(' ');
  }

```

仅支持有效的 scope，这里以微博 OAuth 的文档为例。

在后台我们应该设置好对应的 scope 参数，使其对应不同的权限，validateScope 是拒绝权限，或者提取出部分支持的权限。

我们实现的版本

```

const VALID_SCOPES = ['r_user', 'r_image']

validateScope(user, client, scope) {
  return scope
    .split(',')
    .filter(
      s => client.scope.indexOf(s) >= 0 && VALID_SCOPES.indexOf(s) >= 0
    )
    .join(',')
}

```

全局支持的参数为 r_user 和 r_image，要求请求的 scope 在 client.scope 和 全局的 scope 内。

而 scope 生效还需要给 authenticate 传递 options.scope 参数。authenticate 是用来守护 API 的工具，egg-oauth2-server 重载了 authenticate，调用它会返回守护 API 的中间件来校验 Token。

7. 实现 verifyScope

官方实现的版本

```

function verifyScope(token, scope) {
  if (!token.scope) {
    return false;
  }
  let requestedScopes = scope.split(' ');
  let authorizedScopes = token.scope.split(' ');
  return requestedScopes.every(s => authorizedScopes.indexOf(s) >= 0);
}

```



我们实现的版本

```

verifyScope(token, scope) {
  const requestedScopes = scope.split(',')
  const authorizedScopes = token.scope.split(',')
  return requestedScopes.every(s => authorizedScopes.indexOf(s) >= 0)
}

```

上面代码去掉了空 `token.scope` 的判断，后续再继续完善。

3.10.2 实现刷新验证码接口

1. 实现 `getRefreshToken`

官方实现的版本

```

function getRefreshToken(refreshToken) {
  // imaginary DB queries
  db.queryRefreshToken({refresh_token: refreshToken})
    .then(function(token) {
      return Promise.all([
        token,
        db.queryClient({id: token.client_id}),
        db.queryUser({id: token.user_id})
      ]);
    })
    .spread(function(token, client, user) {
      return {
        refreshToken: token.refresh_token,
        refreshTokenExpiresAt: token.expires_at,
        scope: token.scope,
        client: client, // with 'id' property
        user: user
      };
    });
}

```

我们实现的版本

```
async getRefreshToken(refreshToken) {
  const token = await Refresh.findByToken(refreshToken.refreshToken)
  const client = await Client.findById(refreshToken.client_id)
  const user = await User.findById(refreshToken.user_id)
  return {
    refreshToken: token.token,
    refreshTokenExpiresAt: token.token_expires_at,
    scope: token.scope,
    client,
    user
  }
}
```

2. 实现 revokeToken

官方实现的版本

```
function revokeToken(token) {
  // imaginary DB queries
  return db.deleteRefreshToken({refresh_token: token.refreshToken})
    .then(function(refreshToken) {
      return !!refreshToken;
    });
}
```

我们实现的版本

```
async revokeToken(token) {
  return Refresh.destroy({
    where: {
      token: token.refreshToken
    }
  })
}
```

3. 实现用户认证

文档中没有说要实现这个接口，但是授权码模式还是要进行用户认证的，所以还要实现这个接口。

官方实现的版本

```
function getUser(username, password) {
  // imaginary DB query
  return db.queryUser({username: username, password: password});
}
```

我们实现的版本

```
async getUser(username, password) {
  return User.Auth(username, password)
}
```

3.10.3 实现 authenticate 所需接口

实现 getAccessToken

为了保护路由，需要通过 `accessToken` 去数据库中查找，并验证有效期，再查询出对应的客户端和用户。

官方实现的版本

```
function getAccessToken(accessToken) {
  // imaginary DB queries
  db.queryAccessToken({access_token: accessToken})
    .then(function(token) {
      return Promise.all([
        token,
        db.queryClient({id: token.client_id}),
        db.queryUser({id: token.user_id})
      ]);
    })
    .spread(function(token, client, user) {
      return {
        accessToken: token.access_token,
        accessTokenExpiresAt: token.expires_at,
        scope: token.scope,
        client: client, // with 'id' property
        user: user
      };
    });
}
```



```

    });
  }

```

我们实现的版本

```

Access.getByToken = token => {
  return Access.findOne({ where: { token } })
}

async getAccessToken(accessToken) {
  const token = await Access.getByToken(accessToken)
  const client = await Client.findById(token.client_id)
  const user = await User.findById(token.user_id)
  return {
    accessToken: token.token,
    accessTokenExpiresAt: token.token_expires_at,
    scope: token.scope,
    client, // with 'id' property
    user
  }
}

```

3.11 完善 OAuth 与测试

3.11.1 发放 Token

1. 暂时关闭 JWT

配置 config.default.js 暂时关闭 JWT，便于测试开发。

```

config.jwt = {
  secret: '123456',
  enable: false,
}

```

2. 添加路由

```
app.all('/token', app.oAuth2Server.token(), ctx => ctx.state.oauth.token)
// 获取 access_token
app.all('/authorize', app.oAuth2Server.authorize()) // 获取授权码
app.all('/authenticate', app.oAuth2Server.authenticate(), ctx => {
  ctx.body = ctx.state.oauth
})
```

Token 是用来发放访问令牌的路由, authorize 是用来获取授权码的路由, 而 authenticate 是登录之后可以访问的路由。

3. 修改 home 控制器

```
'use strict'
```

```
class HomeController extends Controller {
  async index() {
    return this.ctx.render('user/login.njk', { query: this.ctx.querystring })
  }
}
```

```
module.exports = HomeController
```

4. 添加视图

views/user/login.njk:

```
{% extends '../layout/app.njk' %}

{% block content %}
<div class="container">
  <h2>login</h2>

  <form action="/authorize?{{query}}" method="post">
    <input type="text" name="username" value="xxx@qq.com">
    <input type="password" name="password" value="888888">
    <button type="submit">login</button>
  </form>
</div>
```

```
{% endblock %}
```

可以提前设置账户和密码，这里用邮箱作为用户名。

5. 添加数据

提前新建用户，记得与上一步的邮箱密码相匹配，连用户都没有是不可能登录成功的，然后在你的数据库 Clients 表中添加数据。

```
client_id: 1,
user_id: 1,
client_secret: 3333,
redirect_uris: http://127.0.0.1:7002/auth/redirect,
grants: password, authorization_code, refresh_token
```

现在数据基本准备完成了。

3.11.2 新建客户端项目

```
mkdir client && cd client && npm init -y
npm install server phin
```

新建 index.js:

```
const server = require('server')
const p = require('phin').promisified

const { get } = server.router
const { redirect, json } = server.reply

const client_id = '1'
const redirect_uri = 'http://127.0.0.1:7002/auth/redirect'
const state = 'somestate'
const client_secret = '3333'

server({ port: 7002 }, [
  get('/', ctx =>
    redirect(
```

```

    `http://127.0.0.1:7001?response_type=code&client_id=${client_id}&redirect_uri=${redirect_uri}&state=${state}`
  ),
  get('/auth/redirect', async ctx => {
    console.log(ctx.query.code)
    const res = await p({
      url: 'http://127.0.0.1:7001/token',
      method: 'POST',
      parse: 'json',
      timeout: 3000,
      form: {
        grant_type: 'authorization_code',
        code: ctx.query.code,
        state: ctx.query.state,
        client_id: client_id,
        client_secret: client_secret,
        redirect_uri: redirect_uri
      }
    })
    console.log(res.body)
    return json(res.body)
  })
})

```

这里使用了 `server` 和 `phin`，一个是 HTTP 服务端，一个是 HTTP 客户端。然后运行 `index.js`，访问 `localhost:7002`。

“/”的作用是跳转到喵览的登录页面，并附上一些数据，这个数据一定要与之前填充到数据库的数据相同。`/auth/redirect` 是用来接收回调的接口，通过 `query` 可以获取授权码，通过 `phin` 这个 HTTP 客户端发起请求可以得到访问令牌。

3.11.3 测试 OAuth

1. 访问 localhost:7002

自行测试的时候可以多注意一下 URL 携带的数据。输入用户名和密码，单击 `login` 进行登录。

2. 自动跳转

自动跳转后的页面如图 3-39 所示。

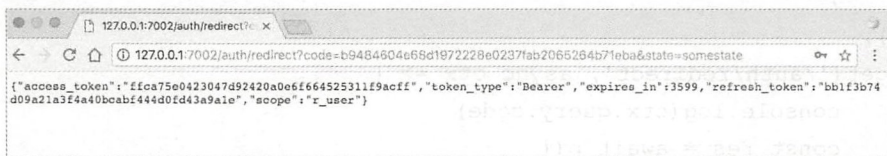


图 3-39

这里的 query 中的 code 就是授权码，我们在客户端里通过 Phin 发起请求，将授权码交换为访问令牌。结果中的 expires_in 是访问令牌的时间，默认是一个小时，而 refresh_token 的有效期默认为两周。

3. PostMan 测试所保护的接口

测试方式如图 3-40 所示。

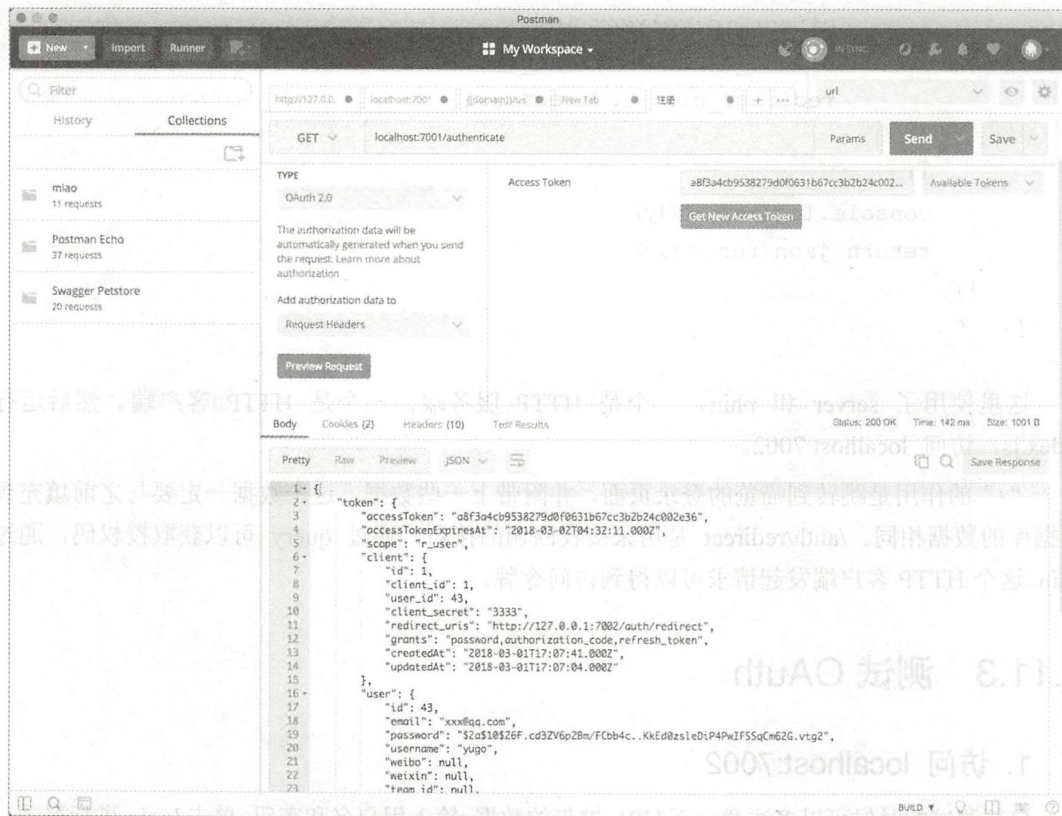


图 3-40

打印出来的数据都是我们每次请求所得到的数据。每次请求都会查询三次数据，如图 3-41 所示。

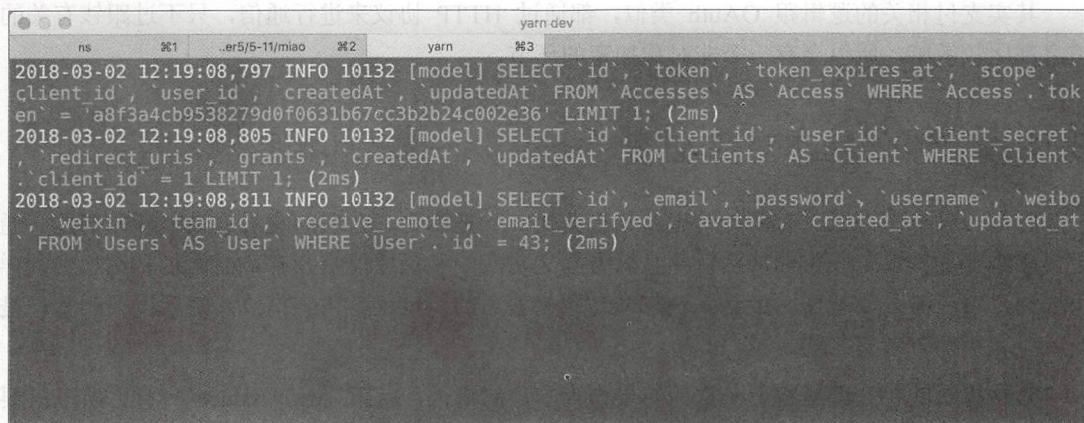


图 3-41

4. 添加 scope

给 router.js 的 authenticate 路由添加域守护:

```
app.all(
  '/authenticate',
  app.oauth2Server.authenticate({
    scope: 'r_user'
  }),
  ctx => {
    ctx.body = ctx.state.oauth
  }
)
```

同样可以。修改为 r_image 就失败了，因为 scope 只有 r_user，这就是对于域的守护，如图 3-42 所示。

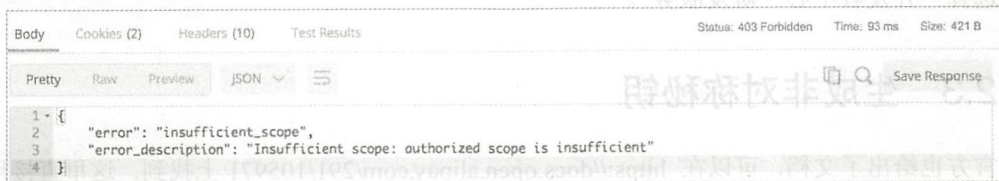


图 3-42

3.12 支付宝支付

其实支付相关的逻辑跟 OAuth 类似，都通过 HTTP 协议来进行通信，只不过跟钱有关的信息都比较敏感，所以会有一个非对称加密的环节。

3.12.1 什么是非对称加密

它是一种加密方式，有两把钥匙，一把公钥、一把私钥。公钥的意思就是公开的钥匙。通过公钥进行加密就是正常的加密，并且只有与之对应的私钥才能解开它。而通过私钥加密同样可以通过对应的公钥解密，但是私钥加密是用来签名的，也就是数字指纹，然后用公钥验证是否是该私钥拥有者签名的。

哈希算法的特性就是将上一次的结果作为本次的输入，就跟 `Array.reduce` 一样，所以只要有任意一个字符被更改，这个哈希值就会不一样。通常签名数据量太大会很慢，而通过哈希算法得到一段哈希值，现在我们只需要签名这个哈希值就可以了。支付宝的支付逻辑就会用到签名，以此来保证是支付宝服务器发送的数据。

非对称加密在其他方面也有应用，比如 HTTPS。有的时候我们希望发送的是密文，可以用私钥对对称密钥进行加密，对称密钥就是只有一把钥匙的加密算法，然后发送给客户端，客户端用这个对称密码与服务器进行密文通信。但是这样比较容易被中间人欺骗，就像突然出现一个中间商来赚差价，要求买方提高价格，要求卖方压低价格。所以要有一个监管机制，那就是 CA，通常操作系统里会内置一些全球可信的证书，这些就是 CA 机构，CA 可以对已公布的公钥和身份信息关联并签名，来保证公钥没有被中间人替代。

3.12.2 注册支付宝

想要上线支付宝的在线支付功能，必须提供营业执照，无论是个体工商户，还是企业都要提供营业执照。对开发者来说，有沙箱环境可供我们进行正常的开发。

进入 <https://openhome.alipay.com> 并进行登录，按照提示，提交个人相关信息成为开发者，然后选择“开发者中心→研发服务”。

3.12.3 生成非对称密钥

官方也给出了文档，可以在 <https://docs.open.alipay.com/291/105971> 上找到。这里直接用命令行生成，使用 OpenSSL 库，OpenSSL 是一个加密解密算法的工具库。Node.js 的 `crypto` 模

块其实就是对 OpenSSL 的封装。

```
openssl genrsa -out sandbox_private_key.pem 2048
```

rsa 就是一种经典的非对称加密算法，基于大数因子分解的数学难题设计，因为难所以才安全。-out 参数指定输出的路径，生成的是私钥，一定要保护好你的私钥。

```
openssl rsa -in sandbox_private_key.pem -pubout -out sandbox_public_key.pem
```

通过私钥生成公钥，把这两个秘钥放到 config/pems 下。然后回到支付宝页面，选择 RSA2 后面的“查看公钥”，单击“修改”按钮，把刚生成的公钥复制进去，如图 3-43 所示。



图 3-43

然后查看支付宝的公钥，新建为 sandbox_public_key.pem 文件，按照上面所生成的公钥的格式进行处理。这里格外要注意的是，我们本地使用的是支付宝的公钥，而不是我们自己的公钥，否则会一直验证失败。

3.12.4 实现

```
npm install alipay-node-sdk
```

1. 添加配置

把配置添加到 config.local.js 中：

```
config.pay = {
  gateway: 'https://openapi.alipaydev.com/gateway.do',
```



```

    appId: '2016072400106012',
    notifyUrl: 'http://5fceb15f.ngrok.io/alipay/callback/',
    returnUrl: 'http://5fceb15f.ngrok.io/alipay/success',
    rsaPrivate: resolve('config/pems/sandbox_private_key.pem'),
    rsaPublic: resolve('config/pems/sandbox_public_key.pem'),
    sandbox: true,
    signType: 'RSA2'
  }
}

```

gateway 和 appId 可以从支付宝开发页面获取，而 notifyUrl 和 returnUrl 稍后会生成，notifyUrl 表示异步接收消息的接口，通常用不到，returnUrl 是支付成功立即返回的接口，也是授权接口。sandbox 表示模拟沙箱环境。

2. 扩展 context

给 extend/context.js 添加逻辑：

```

const payK = Symbol.for('MY_PAY')
const Alipay = require('alipay-node-sdk')

get pay() {
  if (this[payK]) {
    return this[payK]
  }
  this[payK] = new Alipay(this.app.config.pay)
  return this[payK]
},

```

3. 创建 Pay 控制器

controller/pay.js:

```

'use strict'

const Controller = require('egg').Controller
const uuid = require('uuid/v1')

class PayController extends Controller {
  async index() {
    const { ctx, app } = this
  }
}

```

```
const month = parseInt(ctx.params.month) || 1
const user_id = ctx.request.query.id

const order = await app.model.Order.create({
  state: 0,
  order_id: uuid(),
  user_id,
  price: month * 10.0
})

const { returnUrl } = app.config.pay

const data = ctx.pay.pagePay({
  subject: '测试商品',
  body: '测试商品描述',
  outTradeId: order.order_id,
  return_url: returnUrl,
  timeout: '10m',
  amount: order.price,
  goodsType: '0',
  qrPayMode: 2
})

const result = await ctx.curl(app.config.pay.gateway, { data })

ctx.set(result.headers)
ctx.status = result.status
ctx.body = result.data
}

async alipay(ctx) {
  return (ctx.body = 'success')
}

async success(ctx) {
  const ok = ctx.pay.signVerify(ctx.request.query) // 确保是支付宝发送过来的
  if (!ok) {
    ctx.status = 400
  }
}
```

```
    return
  }
  info(ctx.request.query)
  info(ctx.request.body)
  const order = await ctx.model.Order.findOne({
    where: {
      order_id: ctx.query.out_trade_no
    }
  })
  // TODO: 添加权限
  console.log(order)
  order.state = 1
  await order.save()
  return (ctx.body = 'success')
}
}

module.exports = PayController
```

在 `index` 方法中提取出月份和用户的 `id`，创建订单表的实例，计算出价格。然后调用 SDK 的 `pagePay` 生成支付参数，向支付宝的服务器发起请求，把结果显示给用户，让用户进行登录并支付。

对于 `success` 值的授权接口，我们通过 `signVerify` 来保证是支付宝发送过来的请求，然后修改订单的状态，如何授权我们暂时不写，到后期再慢慢地把逻辑提取到 `service` 中。

3.12.5 添加路由

```
router.get('/alipay/pay/:month', controller.pay.index)
router.post('/alipay/callback', controller.pay.alipay) // 异步回调接口
router.get('/alipay/success', controller.pay.success) // 同步授权接口
```

3.12.6 内网穿透

内网穿透其实有非常多的方式，考虑到大家可能没有在线的服务器，所以还是使用 `ngrok` 来实现。

首先到 <https://ngrok.com/> 上注册一个账号，或者使用 `GitHub` 登录，进去之后会看到一个

authtoken 命令，并附有验证码，如下：

```
./ngrok authtoken xxxxxxxxxxxxxxxxxxxxxxx
```

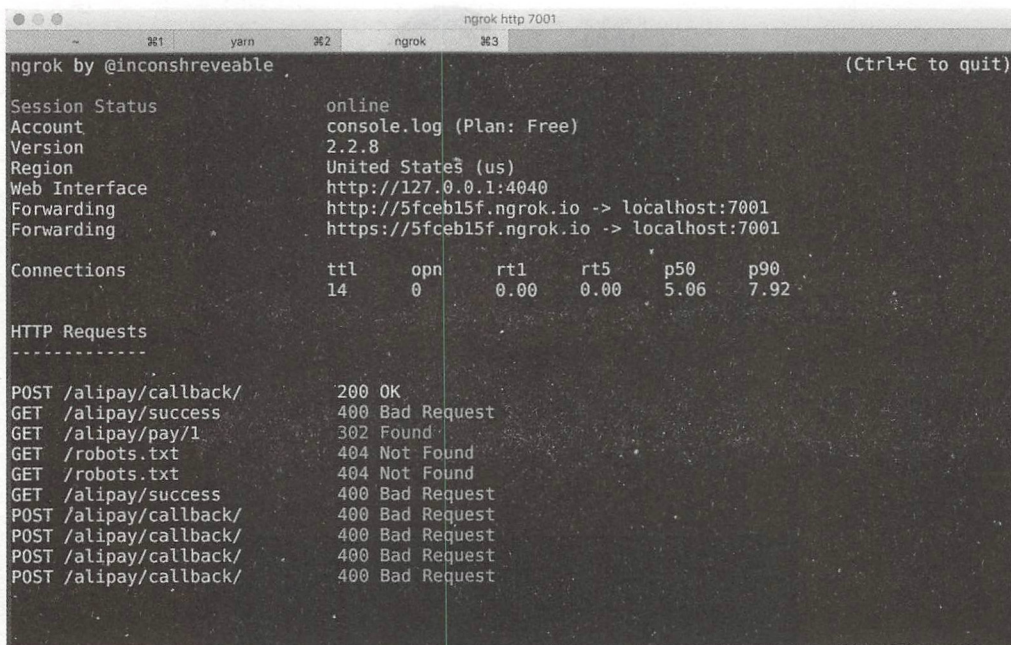
然后可以直接通过 brew 下载 ngrok。

```
brew cask install ngrok
```

下载完成之后，再运行上面的 authtoken 命令。

```
ngrok http 7001
```

运行上面的命令就可以看到与图 3-44 类似的界面。



```
ngrok by @inconshreveable (Ctrl+C to quit)

Session Status      online
Account             console.log (Plan: Free)
Version             2.2.8
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://5fceb15f.ngrok.io -> localhost:7001
                    https://5fceb15f.ngrok.io -> localhost:7001

Connections
  ttl    opn    rt1    rt5    p50    p90
   14     0     0.00   0.00   5.06   7.92

HTTP Requests
-----
POST /alipay/callback/ 200 OK
GET /alipay/success     400 Bad Request
GET /alipay/pay/1       302 Found
GET /robots.txt          404 Not Found
GET /robots.txt          404 Not Found
GET /alipay/success     400 Bad Request
POST /alipay/callback/ 400 Bad Request
POST /alipay/callback/ 400 Bad Request
POST /alipay/callback/ 400 Bad Request
POST /alipay/callback/ 400 Bad Request
```

图 3-44

```
Forwarding http://5fceb15f.ngrok.io -> localhost:7001
```

```
Forwarding https://5fceb15f.ngrok.io -> localhost:7001
```

注意转发的地址，用这个地址替换 config.local.js 里的地址，支付宝页面的地址也要进行修改。

3.12.7 测试

访问支付页面：

`http://5fceb15f.ngrok.io/alipay/pay/1`

单击“使用 PC 支付”按钮，然后单击“登录账户付款”按钮，如图 3-45 所示。



图 3-45

沙箱的账户可以在支付宝开发的页面找到，然后进行登录与支付。

登录成功后，输入支付密码，继续进行支付过程示意图略。

查看数据状态：

```
GET /alipay/success 200 OK
POST /alipay/callback/ 200 OK
GET /alipay/pay/1 302 Found
GET /alipay/success 400 Bad Request
GET /robots.txt 404 Not Found
GET /robots.txt 404 Not Found
GET /alipay/success 400 Bad Request
GET /alipay/success 400 Bad Request
GET /alipay/success 400 Bad Request
GET /alipay/success 400 Bad Request
GET /favicon.ico 200 OK
```

13	d1e9e6c0-1dea-11e8-86fa-ab3ccd729d9a	NULL	10	1	2018-03-02 07:25:01	2018-03-02 07:25:51
14	c7b8c5b0-1ded-11e8-86fa-ab3ccd729d9a	NULL	10	0	2018-03-02 07:46:13	2018-03-02 07:46:13
15	ccc99de0-1ded-11e8-86fa-ab3ccd729d9a	NULL	10	1	2018-03-02 07:46:21	2018-03-02 07:49:49
16	85f50750-1dee-11e8-812f-8dbdd06937b9	NULL	10	0	2018-03-02 07:51:32	2018-03-02 07:51:32
17	ac8257e0-1df0-11e8-ba93-03072e5ff797	NULL	10	1	2018-03-02 08:06:55	2018-03-02 08:07:40

可以看到最后的 `success` 是 200，最后一条订单的状态是 1，这里格外要注意使用公钥的问题，一定要是支付宝的公钥。

3.13 社会化登录

1. 创建记录表

```
npx sequelize model:create --name auth --attributes provider:string,
uid:string,user_id:integer
```

2. 修改 migration

```
'use strict'
module.exports = {
  up(queryInterface, Sequelize) {
    return queryInterface.createTable('auths', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      provider: {
        type: Sequelize.STRING
      },
      uid: {
        type: DataTypes.STRING
      },
      user_id: {
        type: Sequelize.INTEGER
      }
    })
  },
  down(queryInterface, Sequelize) {
    return queryInterface.dropTable('auths')
  }
}
```

3. model

```
'use strict'

module.exports = function({ model: sequelize, DataTypes }) {
  const Auth = sequelize.define(
    'Auth',
    {
      provider: DataTypes.STRING,
      uid: DataTypes.STRING,
      user_id: DataTypes.INTEGER
    },
    { timestamps: false }
  )
  return Auth
}
```

关闭时间戳。

4. 同步到数据库表中

```
npx sequelize db:migrate
```

```
> npx sequelize db:migrate
Sequelize [Node: 8.9.0, CLI: 2.8.0, ORM: 4.33.4]
Loaded configuration file "config/config.json".
Using environment "development".
== 20180302082321-create-auth: migrating =====
== 20180302082321-create-auth: migrated (0.457s)
```

5. 安装插件

```
npm i egg-passport-github
```

```
exports.passportGithub = {
  enable: true,
  package: 'egg-passport-github'
}
```

```
config.passportGithub = {
  key: '9c29ac4879f6d947bf09',
```

```
secret: 'd05a879ca3ba7d1386b65351d6e8844370cea9f7'
}
```

key 和 secret 在个人设置里的 OAuth 设置中申请得到。

记得设置 callback，格式是你的域名加上 /passport/github/callback，稍后这个域名会由 ngrok 生成。

6. 配置路由

```
mount(['local', 'github'], app.passport, controller)

router.get('/passport/github', controller.passport.github)
router.get('/passport/github/callback', controller.passport.github)
```

记得要在 mount 之后再注册路由。

7. 添加逻辑

新建 passport/github.js 文件：

```
'use strict'

const R = require('ramda')
module.exports = async (ctx, user) => {
  const data = { uid: user.id, provider: user.provider }
  const auth = (await ctx.model.Auth.findOrCreate({
    where: data,
    default: data
  }))[0]

  if (auth.user_id) {
    const existsUser = await ctx.model.User.findOne({
      where: { id: auth.user_id }
    })
    const raw_user = R.omit(['password'], existsUser.toJSON())
    const token = await ctx.sign_token(raw_user)
    ctx.body = token
    return token
  }
```



```
// 调用 service 注册新用户
const newUser = await ctx.model.User.create({
  username: user.displayName,
  avatar: user.photo,
  email: user.profile.emails[0].value
})
auth.user_id = newUser.id
await auth.save()
const raw_user = R.omit(['password'], newUser.toJSON())
const token = await ctx.sign_token(raw_user)
ctx.body = token
return token
}
```

通常 GitHub 登录之后，会在 Auth 中创建一条数据，provider 表示登录的方式，比如 QQ、GitHub 等，而 uid 则是 GitHub 所提供的用户 ID，user_id 则是对应我们自己系统的 user_id。首先从 Auth 里找有没有对应本系统的用户，没有则会创建，但是创建的用户是没有用户密码和激活的，所以第一次通过 GitHub 登录时，还应该在前端做一些处理，这里暂不处理。

8. 测试

ngrok http 7001

开启内网穿透并得到域名后，别忘记修改 GitHub OAuth 回调的地址。

```
ngrok by @inconshreveable

Session Status      online
Account             console.log (Plan: Free)
Version             2.2.8
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://f98ad6ed.ngrok.io -> localhost:7001
Forwarding           https://f98ad6ed.ngrok.io -> localhost:7001

Connections         ttl      opn      rt1      rt5      p50      p90
0                  0        0.00     0.00     0.00     0.00
```

这里笔者的回调地址是 <http://f98ad6ed.ngrok.io>，首先访问 <http://f98ad6ed.ngrok.io/passport/github>，如图 3-46 所示。

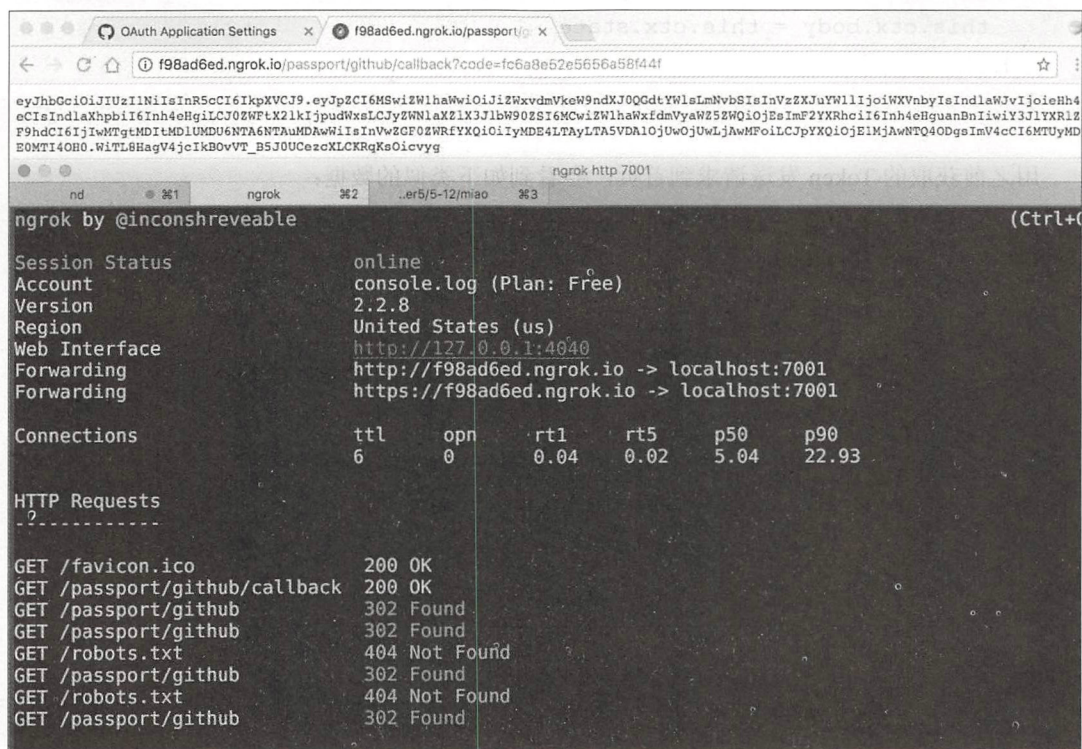


图 3-46

因为笔者已经登录并授权过了，所以就直接得到了 Token。

9. 开启 JWT

配置 config/config.default.js:

```
config.jwt = {
  secret: '123456',
  enable: true,
  ignore: [/\/passport/i, /\/api/]
}
```

临时修改 HomeController，用来查看当前 JWT 签名数据的信息。

```
class HomeController extends Controller {
  async index() {
    // return this.ctx.render('user/login.njk', { query: this.ctx.querystring })
  }
}
```

```
    this.ctx.body = this.ctx.state
  }
}
```

用之前获取的 Token 发送请求到首页，会看到如下类似的数据：

```
1- {
2-   "user": {
3-     "id": 46,
4-     "email": null,
5-     "username": "console.log",
6-     "weibo": null,
7-     "weixin": null,
8-     "team_id": null,
9-     "receive_remote": 0,
10-    "email_verified": 0,
11-    "avatar": "https://avatars3.githubusercontent.com/u/10082395?v=4",
12-    "created_at": "2018-03-02T09:11:22.000Z",
13-    "updated_at": "2018-03-02T09:11:22.000Z",
14-    "iat": 1520055837,
15-    "exp": 1520142237
16-  }
17- }
```


4 chapter

第 4 章 构建后台管理页面

4.1 后端开发

4.1.1 安装 VSCode 插件

Vue 官方对于 VSCode 开发了专门的插件，在插件里搜索 Vetur，然后进行安装，假如你发现 Vue 文件没有高亮，那么可能是安装文件发生了丢失，删除插件，退出 VSCode 后重新安装即可。

如何打开调试

单击“帮助→切换开发人员工具”选项，打开控制台，可以看到调试信息，假如发现丢失 xxx 模块，就说明安装的时候丢失了东西。可以尝试重装，也可以进入 `~/.vscode/extensions` 对应的插件下，输入 `npm install` 安装依赖即可。

4.1.2 获取脚手架

- 下载起始源码

```
git clone https://github.com/octref/veturpack.git
```


- 安装依赖

```
cd veturpack && npm install
```

这个脚手架使用 Vue 框架，Vue 是一个 MVVM 框架，即数据与视图同步框架，笔者也录制了好几个系列的 Vue 免费教程，都可以在 <https://nodelover.me/course/vue-basic> 找到，没有基础的读者可以先观看这些入门教程视频。

4.1.3 安装依赖

在开发过程中需要与后端进行通信，我们使用前后端一致的 API axios 库，以及帮助函数库 ramda。jQuery 是稍后用来测试 JSONP 请求的，只做演示，不在实际项目中使用，测试完可以自行删除，而 @types 文件可以让 VSCode 提供 jQuery 的代码提示，base-64 是对密码进行编码的工具。

```
npm install ramda axios jquery base-64 -S
npm install @types/jquery -D
```

4.1.4 修改代码

我们需要修改一下代码。

添加主题样式

编辑 build/index.html，添加 bulma 主题与一些样式，样式可以在 <https://jenil.github.io/bulmaswatch/> 中进行挑选。

```
<link href="https://cdn.bootcss.com/bulma/0.6.2/css/bulma.min.css" rel="stylesheet">
<link href="https://cdn.bootcss.com/bulmaswatch/0.6.2/sandstone/bulmaswatch.min.css" rel="stylesheet">
```

编辑 .eslintrc.json，默认是没有添加全局变量的，所以有的时候可能会报错，添加代码如下。

```
"globals": {
  "process": true,
  "module": true,
```

```
"exports": true,  
"require": true  
}
```

4.1.5 跨域请求

1. 什么是跨域

有以下任意情况都禁止通信。这是由浏览器的安全策略导致的，其实请求与结果是正常发起的，但是容易导致安全问题，所以浏览器会忽略服务返回的结果，而抛出错误。

- 协议不同，如 HTTP、HTTPS；
- 端口不同，如 localhost:4000 与 localhost:7001；
- 主域相同，子域不同，如 a.hello.com 与 b.hello.com；
- 主域不同，如 taobao.com 与 baidu.com；
- IP 地址和域名之间也算是跨域。

2. 如何解决跨域

基本都需要在服务端做一些处理。这里只阐述一些 JSON 与 CORS，还有一些其他的方法，但是更加冷门。

JSONP

浏览器的 XHR 请求是有跨域的安全策略，但是 script 标签发送的请求是不受限制的，所以就有人想出了非常规的方式，通过 createElement 创建一个 script dom，指定它的 src 去请求某个地址，然后返回一个 JS 代码，JS 代码中有一个函数，调用函数可以得到数据。这种方式也有一些局限，比如无法添加 Header，也就无法完成验证等，因为请求是由 script dom 发起的，所以没办法做自定义，这也是前几年比较常用的方式。

CORS

这是当前最常用的方式，大多数浏览器都已经支持。其实就是在 HTTP 请求头和返回头添加一些参数让浏览器识别。

CORS 对于简单的 GET、HEAD 与 POST 会发起一次请求，而对于复杂的请求，比如添加了一些自定义的请求头或者其他方法（如 PUT、DELETE 等），那么会发起两次请求，第一次是 OPTIONS 请求，主要用于向服务器询问，是否支持方法与请求头。

在服务器中设置返回的头字段，例如，Access-Control-Allow-Origin: http://foo.example，代

表来自 foo.example 的请求，可以跨域访问。也可以设置为*，即允许所有。

当然还有一些其他的，如下：

- Access-Control-Allow-Methods: POST, GET, OPTIONS;
- Access-Control-Allow-Headers: X-PINGOTHER, Content-Type;
- Access-Control-Max-Age: 86400。

第一条表示接收 POST、GET 和 OPTIONS 请求，第二条表示所接收的请求头字段，第三条表示有效期时间。

4.1.6 修改后端代码支持跨域

Egg.js 早已经为我们提供了这些组件。

1. 安装 egg-cors

```
npm i egg-cors -S
```

修改 plugin.js:

```
exports.cors = {  
  enable: true,  
  package: 'egg-cors'  
}
```

修改 config.default.js:

```
config.security = {  
  csrf: { ignore: () => true, ignoreJSON: true },  
  domainWhiteList: ['http://localhost:4000']  
}
```

表示允许来自 http://localhost:4000 端口的跨域请求。

2. 添加 jsonp 支持

修改 config.local.js:

```
config.jsonp = { whiteList: ['localhost'] }
```

添加 jsonp 路由。

```
router.get(
  '/api/v1/jsonp',
  app.jsonp(),
  (ctx, next) => {
    if (ctx.query.username !== 'dd' && ctx.query.password !== '8888888') {
      return ctx.throw(403)
    }
    return next()
  },
  ctx => (ctx.body = { name: '123' })
)
```

jsonp 请求都需要用 jsonp 中间件进行处理，因为没办法在 header 中验证，所以只好放到 query 中。

3. 添加计算表数据的 API

调用模型的 count 方法就可以获取数据量。

```
const {
  forEachObjIndexed,
  forEach,
  keys,
  values,
  pipe,
  map,
  zipObj
} = require('ramda')
```

```
const R = require('ramda')
```

```
async function getModelCount(ctx) {
  // * v1 错误示例
  // const count = {}
  // forEachObjIndexed(async (model, modelName) => {
  //   count[modelName] = await model.count()
  // }, ctx.app.model.models)
```



```

// * v2
// const models = ctx.app.model.models
// const names = keys(models)
// const toCountPromises = map(m => m.count(), values(models))
// const _values = await Promise.all(toCountPromises)
// ctx.body = zipObj(names, _values)

// * v3
const models = ctx.app.model.models
const getValuePromises = R.pipe(R.values, R.map(R.invoker(0, 'count')))
const names = keys(models)
const values = await Promise.all(getValuePromises(models))
ctx.body = R.zipObj(names, values)
}

module.exports = {
  getModelCount
}

```

这里写了 3 种方式，第一种直接用 `forEachObjIndexed`，因为 `model.count` 方法返回的是 `Promise`，所以获取数据要用 `await`，用 `await` 就必须要用 `async` 函数。这个 `async` 是回调上面的函数，这样容易出现 `count` 为空的情况，因为回调的 `async` 函数并没有进行等待。这也是初学者的误区。

第二种就是通过 `ramda` 提供的 `keys` 和 `values` 获取对象的属性数组与值数组。值数组就是模型的数组，通过 `map` 调用模型的静态 `count` 方法，这样它就是一个 `Promise` 数组了，通过 `Promise.all` 把一个 `Promise` 数组进行同步调用，然后通过 `await` 进行等待，获取结果数组。通过 `zipObj` 将 `names` 和 `values` 重新组合为对象，即将 `names` 数组中的值作为 `key`，将 `values` 中的值作为 `value`。

第三种就是通过 `R.pipe` 进行串联调用，`invoker(0, 'count')` 跟 `m => m.count()` 相等，`0` 表示接收的参数。即将 `models` 传递给 `R.values` 之后，得到一个数组，再传给 `R.map` 对数组的每一项调用 `count`。最后通过 `Promise.all` 进行同步调用。

别忘记导出并修改 `app/router.js`：

```

const {
  getModelCount: countRouter

```



```
} = require('./util')
```

```
router.get(
  '/api/v1/admin_count',
  auth({
    name: 'dd',
    pass: '888888'
  }),
  countRouter
)
```

4. 修改前端的代码

新建 client/api.js:

```
import axios from 'axios'
```

```
const baseURL = 'http://localhost:7001/api/v1/admin'
```

```
export function getCount () {
  return axios.get(baseURL + '_count',{
    auth: {
      username: 'dd',
      password: '888888'
    }
  })
}
```

修改 client/app.js。

将 API 挂载到 Vue 实例上，便于我们随处调用。

```
import * as api from './api'

Vue.prototype.$api = api
```

修改 client/views/home.vue。

添加一个 Vue 挂载的时候会触发的函数，即 mounted:

```
import $ from 'jquery'
```



```
async mounted() {
  try {
    const counts = await this.$api.getCount()
    console.log(counts.data)
    $.ajax('http://127.0.0.1:7001/api/v1/jsonp', {
      dataType: 'jsonp',
      data: {
        username: 'dd',
        password: '888888'
      }
    }).then(console.log)
  } catch (e) {
    console.error(e)
  }
}
```

5. 启动前端的服务

```
npm run dev
```

打开 localhost:4000，然后打开控制台就可以看到正常的结果。

```
Home.vue?6dc74
▶ {Access: 21, Auth: 1, Authorization: 30, Category: 0, Client: 1, ...}
▶ {name: "123"} "success"
jquery.i
```

6. 添加获取后端表结构的路由

```
function mapToInputType(type) {
  const _maps = { integer: 'number', string: 'text', text: 'textarea' }
  return _maps[type] || type
}

function getModelConfig(ctx) {
  const config = {}
  forEachObjIndexed((model, modelName) => {
    config[modelName] = {}
    forEachObjIndexed((attr, attrName) => {
      config[modelName][attrName] = mapToInputType(attr.type.key.toLowerCase())
    })
  })
}
```



```
    }, model.tableAttributes)
  }, ctx.app.model.models)
ctx.body = config
// ! 未优化的方法
// ctx.body = Object.keys(ctx.app.model.models).reduce((acc, modelName) => {
//   const currentModel = ctx.app.model.models[modelName]
//   const currentModelAttrs = currentModel.tableAttributes
//   const modelValue = Object.keys(currentModelAttrs).reduce((acc, attr) => {
//     const currentModelAttrInstance = currentModel.tableAttributes[attr]
//     const type = currentModelAttrInstance.type.key.toLowerCase()
//     acc[attr] = mapToInputType(type)
//     return acc /* 将复杂的对象转化为 { image_name: string }
//   }, {})
//   acc[modelName] = modelValue
//   return acc
// }, {})
}

module.exports = {
  getModelConfig
}
```

从模型 `model.tableAttributes` 可以获取 `table` 字段的相关信息。由于它也是对象，所以还需要进行一次对象遍历，`type.key` 就是它的类型，不过是大写的，`toLowerCase` 将其转化为小写，然后通过 `mapToInputType` 做一些修改映射，即 `integer` 变成 `number`，因为作为 `input type` 的时候，需要让 HTML 识别。

有一个未优化的版本，使用的都是原生的 API，会稍微复杂一些，不过还是有一些优化的空间，即把相同的 `reduce` 提取出来。

添加路由，修改 `router.js`：

```
const {
  getModelCount: countRouter,
  getModelConfig: adminRouter
} = require('./util')

router.get(
  '/api/v1/admin_table',
```




```
auth({
  name: 'dd',
  pass: '8888888'
}),
adminRouter
)
```

使用 postman 测试该接口，可以得到以下类似的数据：

```
1- {
2-   "Access": {
3-     "id": "number",
4-     "token": "text",
5-     "token_expires_at": "date",
6-     "scope": "text",
7-     "client_id": "number",
8-     "user_id": "number",
9-     "createdAt": "date",
10-    "updatedAt": "date"
11-  },
12-  "Auth": {
13-    "id": "number",
14-    "provider": "text",
15-    "uid": "text",
16-    "user_id": "number"
17-  },
18-  "Authorization": {
19-    "id": "number",
20-    "code": "text",
21-    "expires_at": "date",
22-    "redirect_uri": "text",
23-    "scope": "text",
24-    "client_id": "text"
```

4.1.7 在前端添加存储

1. 修改 client/api.js

将获取的数据保存到全局状态中：

```
import axios from 'axios'

const baseURL = 'http://localhost:7001/api/v1/admin'

export function getConfig () {
  return axios.get(baseURL + '_table',{
    auth: {
```



```
    username: 'dd',
    password: '8888888'
  }
})
}

export function getCount () {
  return axios.get(baseUrl + '_count',{
    auth: {
      username: 'dd',
      password: '8888888'
    }
  })
}
```

2. 修改 store/index.js

修改并添加以下代码，不要把已有的代码全部删除：

```
const state = {
  tables: {},
  counts: {}
}

const set_helper = prop => (state, payload) => (state[prop] = payload[prop])

const mutations = {
  SET_TABLES: set_helper('tables'),
  SET_COUNT: set_helper('counts')
}
```

3. 修改 views/home.vue

```
async mounted() {
  const tables = await this.$api.getConfig()
  const counts = await this.$api.getCount()
  this.$store.commit('SET_TABLES', { tables: tables.data })
  this.$store.commit('SET_COUNT', { counts: counts.data })
}
```



4.2 模型列表

首先从最简单的做起，第一期把 CURD 做好就可以了，这一节先把所有的数据都列出来。

1. 修改后台查询接口

修改 `app/controller/admin.js`:

```
/**
 * get list
 * @param {object} ctx Context
 */
async index({ query }) {
  R.mapObjIndexed((val, key) => {
    try {
      if (parseInt(val)) {
        query[key] = parseInt(val)
      }
    } catch (e) {
      return
    }
  }, query)
  const data = await this.model.findAll(query)
  this.normarlize(data)
}
```

复写所继承的 `index` 即可，这里需要把 `query` 对象做一下转义，通过 HTTP 传输过来的数据经过 `query` 解析后，会变成字符串，构建 SQL 语句的时候就会出错，所以要把它转换成数字。

2. 修改 `eslint` 配置

我们要用到 `async` 语法，其实在 `vue-app` `babel` 插件里就已经处理过 `async` 语法了，但是 `eslint` 并没有配置，所以会造成报错。修改一下 `ESLint` 解析的版本：

```
"parserOptions": {
  "ecmaVersion": 2018
},
"env": { "node": true, "es6": true }
```



3. 添加帮助方法

新建 client/utils.js:

```
import { forEachObjIndexed } from 'ramda'
```

```
export const mapObjToArray = (fn, obj) => {  
  const arr = []  
  forEachObjIndexed((val, key) => {  
    arr.push(fn(val, key, obj))  
  }, obj)  
  return arr  
}
```

```
export const mapObj = (fn, obj) => {  
  const _obj = {}  
  forEachObjIndexed((val, key) => {  
    const _ret = fn(val, key, obj)  
    obj[key] = _ret  
  }, obj)  
  return _obj  
}
```

添加两个帮助方法，便于后面调用，这些都是遍历对象的方法，一个是返回对象，一个是返回数组。

4. 在入口添加载入数据逻辑

在 client/app.js 里添加代码:

```
import { mapObjToArray } from './utils'  
import CreateListView from './views/CreateListView'
```

```
~(async () => {  
  const { data: tables } = await api.getConfig()  
  const { data: counts } = await api.getCount()  
  store.commit('SET_COUNT', { counts })  
  const routes = mapObjToArray(  

```




```
(struct, name) => ({ path: '/model/' + name + '/:page?', component:
CreateListView(name) })),
  tables
)
router.addRoutes(routes)
})()
```

之所以要在入口处添加载入数据的代码，是因为可以在每一次刷新的时候都能获取数据，假如还是在 Home.vue 的 mounted 生命周期函数中调用获取数据，当 URL 跳转到某一个页面，而不是在 Home.vue 的时候，刷新页面就执行不到获取数据的逻辑。这里不做缓存的原因是后台的访问量本来就小，而且实时性要求高，没有必要缓存数据。

CreateListView 是我们稍后要实现的方法，它可以动态地创建页面，为每一个模型都创建一个页面，与对应的 path 相互组合，推入 routes 数组中，最后通过 addRoutes 动态地添加路由。

```
async function someFunction(){
}
~(someFunction)()
```

最上面的代码和这里的代码功能其实一样，都是执行一段函数中的代码，只不过最上面用的是匿名函数，之所以在前面用一个~，是为了让浏览器知道这是一个表达式，这个符号也可以是其他的数学符号，比如+、-，并且这种函数也有一个名字叫作 IIFE，即自执行函数。

5. 修改样式

修改 components/App.vue 中的样式：

```
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'avenir next', avenir,
helvetica, 'helvetica neue', Ubuntu, 'segoe ui',
  arial, sans-serif;
}

#app {
  padding: 20px;
}
```



6. 创建视图工厂

新建 `views/CreateListView.js`:

```
import ModelList from './ModelList.vue'

export default function createListView (type) {
  return {
    name: `${type}-list-view`,

    render (h) {
      return h(ModelList, { props: { type } })
    }
  }
}
```

这个文件只是将 `ModelList.vue` 做一层包裹，它就像一个工厂，生成添加了一个 `name`，以及传递的 `type` 的视图。这个 `type` 其实就是模型的名字，比如 `User`、`Order` 等。

这里的组件是通过 `render` 函数进行渲染的，其实我们之前写的 `template` 中的代码最终都会被编译成 `render` 函数，像这种没有 `data` 数据的组件，现在我们直接写反而更简单。`h` 其实就是渲染函数，第一个参数是渲染的对象，第二个参数是配置项，比如说传递给渲染对象的属性 `props`。

创建视图模板

先安装一个依赖，用来格式化 JSON 数据，其实一些成熟的后台系统操作的也是这些 JSON，样式之类先不着急，先把逻辑业务走通再谈其他方面。

```
npm install -S json-pretty-html
```

创建 `views/ModelList.vue`。

- 添加样式

```
<style>
.json {
  font-family: Menlo, Monaco, 'Courier New', monospace;
  font-weight: normal;
  font-size: 14px;
```

```
line-height: 16px;
letter-spacing: 0;
background-color: #24282a;
color: #d4d4d4;
text-align: left;
border-top: 1px solid #121516;
padding-top: 10px;
padding-bottom: 10px;
margin: 0;
}
.json-pretty {
padding-left: 30px;
padding-right: 30px;
}
.json-selected {
background-color: rgba(139, 191, 228, 0.19999999999999996);
}
.json-string {
color: #6caedd;
}
.json-key {
color: #ec5f67;
}
.json-boolean {
color: #99c794;
}
.json-number {
color: #99c794;
}
</style>
```

上面的代码就是用来给 JSON 化后的样式添加一些颜色，这些样式都是从 json-pretty-html 库里复制过来的。

- 添加逻辑

后台页面最基础的功能就是分页，现在我们先实现最简单的分页，因为之前做好了作为 where 条件的 query，现在我们可以非常方便地随意定制数据。导入 mapState、pretty 及 ramda 函数库，mapState 把全局状态里存储的表信息放到本组件内，pretty 是格式化代码，而 ramda 则是帮助库。

这个组件与工厂函数一样也需要一个 type 参数，因为可以设置分页，所以 data 上要有一个 limit 来控制每页显示的数量，以及 displayedItems 显示的内容。

当前的页数是通过路由的参数来获取的，默认是第一页，而最大页面数 maxPage 则是通过全局状态里的记录条数来计算得到的。这里要对 page 进行监听，因为当从 /model/User/1 改变到 /model/User/2 的时候，并不会重新渲染组件，所以不会触发生命周期函数，这是初学者比较容易犯的一个错误，所以我们通过监听 page 来实现获取新数据。

在 methods 中挂载 pretty 是因为 Vue 只能获取实例中的数据和 method，添加一个 loadItems 方法来刷新数据，它接收一个 to 与 from，to 是当前页。获取到数据之后，赋值给 displayedItems。当然在 mounted 的时候也要调用一次 loadItems 方法。

```
<script>
import { mapState } from 'vuex'
import * as R from 'ramda'
import pretty from 'json-pretty-html'

export default {
  name: 'model-list',
  props: {
    type: String
  },
  data() {
    return {
      limit: 20,
      displayedItems: []
    }
  },
  computed: {
    page() {
      return Number(this.$route.params.page) || 1
    },
    maxPage() {
```




```

const { counts } = this.$store.state
return Math.ceil(counts[this.type] / this.limit) || 1
},
hasMore() {
  return this.page < this.maxPage
}
},
watch: {
  page(to, from) {
    this.loadItems(to, from)
  }
},
methods: {
  pretty,
  async loadItems(to = this.page, from = -1) {
    if (this.page < 0 || this.page > this.maxPage) {
      this.$router.replace(`/model/${this.type}/1`)
      return
    }
    this.displayedPage = to
    const where = {
      limit: this.limit
    }
    if (this.limit * (this.page - 1)) {
      where.offset = this.limit * (this.page - 1)
    }
    let { data } = await this.$api.getList(this.type, where)
    console.log(data)
    this.displayedItems = data
  }
},
async mounted() {
  await this.loadItems()
}
}
</script>

```



7. 添加获取数据的方法

上一节的 `getList` 方法我们还没有实现，现在来实现它。

在 `client/api.js` 中添加如下代码：

```
const fetch = axios.create({
  baseURL,
  auth: {
    username: 'dd',
    password: '8888888'
  }
})

export function getList (model, query) {
  return fetch.get(model, {
    params: query
  })
}
```

通过 `create` 方法创建一个可以复用的实例。

8. 在首页添加一些链接

修改 `views/Home.vue`，添加模板：

```
<template>
  <div class="page">
    <div class="container">
      <h2>Model List</h2>
      <template v-for="(_, name) in tables">
        <div :key="name" class="is-size-4">
          <router-link class="tag is-small" :to="{ path: '/model/' + name }"
v-text="name"/>
        </div>
      </template>
    </div>
  </div>
</template>
```



添加逻辑：

```
<script>
import { mapState } from 'vuex'

export default {
  computed: {
    ...mapState(['tables'])
  },
}
</script>
```

从 vuex 里通过 mapState 获取 tables 数据，通过 for 遍历它，我们不关心其中的字段，只想要获取 name 就行，把 name 与 URL 进行拼接即可。

9. 删除原有的样式

在 client/components/App.vue 里还存在一些样式，我们修改一下，去掉不必要的部分。

```
body {
  margin: 0;
}

#app {
  padding: 20px;
}
```

10. 测试页面

现在列表页面就已经基本完成了，我们来测试一下。

打开 localhost:4000 即可看到链接列表。

单击其中的 Authorization 项，可以查看数据，如图 4-1 所示。





图 4-1

假如发现某些属性传递不对，则要注意传递给组件的语法 `:name="tableName"` 一定不要忘记冒号，少了冒号就不是变量了，而是字符串 `tableName`。

4.3 添加数据

由于每一个模型的表单都不一样，所以我们需要动态地创建表单，像 React 那样，可以直接写 JS 逻辑的模型反而会更方便一些。其实 Vue 也是支持的，Vue 的 `template` 会编译成 `render` 函数，现在我们自己写 `render` 函数就好了，不过为了更加方便书写，会加上 `jsx` 插件的支持。

1. jsx 语法

以下两种方式等价，在了解 `jsx` 之前，我们先了解一下前置概念。其实渲染函数就是构建 JS 树对象，即虚拟 DOM，用 JS 对象来模拟 DOM，但是这个 DOM 并没有实际渲染到页面上，因为渲染到页面上比较消耗性能。通过修改这个 JS 树对象来实现增量更新页面。相信读者或多或少都玩过一些游戏，增量更新并不是非常新的技术，Git 协议其实也可以看作增量更新。更新 JS 树对象，即虚拟 DOM。将新 JS 树对象与老 JS 树对象进行对比，找出需要更新的部分，



然后反应到 DOM 上即可。

那么如何构建 JS 树对象呢？其实就是通过 Vue 中 `h` 函数的调用，在 React 里叫 `createElement`。第一个参数是标签的名字，比如 `div`，第二个参数是属性对象，第三个参数是子孙节点。通过 `jsx`，我们就可以像写 HTML 那样来声明 JS 树对象，这样可以降低理解难度，将函数调用转化为标签声明。

Vue `jsx` 的更多信息可以在 <https://github.com/vuejs/babel-plugin-transform-vue-jsx#usage> 找到，跟 React 的 `jsx` 还是有一些不同的。

```
render(h) {  
  return h('div', {  
    props: {  
      msg: 'hi'  
    },  
    attrs: {  
      id: 'foo'  
    },  
    domProps: {  
      innerHTML: 'bar'  
    },  
    on: {  
      click: this.clickHandler  
    },  
    nativeOn: {  
      click: this.nativeClickHandler  
    },  
    class: {  
      foo: true,  
      bar: false  
    },  
    style: {  
      color: 'red',  
      fontSize: '14px'  
    },  
    key: 'key',  
    ref: 'ref',  
    refInFor: true,  
    slot: 'slot'
```



```

    ))
  }
  render (h) {
    return (
      <div
        id="foo"
        domPropsInnerHTML="bar"
        onClick={this.clickHandler}
        nativeOnClick={this.nativeClickHandler}
        class={{ foo: true, bar: false }}
        style={{ color: 'red', fontSize: '14px' }}
        key="key"
        ref="ref"
        refInFor
        slot="slot">
      </div>
    )
  }
}

```

2. 安装依赖

```
npm install vue-modalтор vue-notifications mini-toastr
```

vue-modalтор 用来弹出表单静态框，而 notifications 用来显示消息是否成功，它依赖 mini-toastr。

3. 初始化插件

```

import VueModalтор from 'vue-modalтор'
import dbFieldMap from './store/map_zh-CN'

Vue.use(VueModalтор)

import VueNotifications from 'vue-notifications'
import miniToastr from 'mini-toastr' // https://github.com/se-panfilov/mini-toastr

miniToastr.init()

```



```

function toast ({ title, message, type, timeout, cb }) {
  return miniToastr[type](message, title, timeout, cb)
}

const options = {
  success: toast,
  error: toast,
  info: toast,
  warn: toast
}

Vue.use(VueNotifications, options)
Vue.prototype.$dbMap = dbFieldMap

~(async () => {
  const { data: tables } = await api.getConfig()
  const { data: counts } = await api.getCount()
  store.commit('SET_COUNTS', { counts })
  store.commit('SET_TABLES', {
    tables
  })
  const routes = mapObjToArray(
    (struct, name) => ({ path: '/model/' + name + '/:page?', component:
CreateListView(name) }),
    tables
  )
  router.addRoutes(routes)
})()

```

关于插件如何配置，笔者都是直接复制插件里的示例代码，我们也不用去管它是什么意思，能用就行。

之前这里的 SET_COUNTS 与 SET_TABLES 不太一致，现在我们统一加上 S，而且之前 SET_TABLES 也没有调用，记得加上，因为稍后在创建表单时我们要用到这里的表单配置。

4. 创建显示名字映射

新建 store/map_zh-CN.js，其中存储的数据是表单显示的名字，这里笔者把所有的字段进行了汇总，假如字段有冲突，则可以加上命名空间，即放到对应的模型名字对象下。



```
export default {
  id: 'ID 号',
  token: '访问令牌',
  token_expires_at: '令牌过期时间',
  scope: '授权范围',
  client_id: '客户端 ID 号',
  user_id: '用户 ID 号',
  createdAt: '创建日期',
  updatedAt: '更新日期',
  code: 'OAuth2 授权码',
  expires_at: '超时时间',
  redirect_uri: '访问令牌安全的发放网址'
}
```

5. 新建 VForm 组件

新建 components/VForm.js。

- 导入依赖

```
import Vue from 'vue'
import { debounce, map, pick, omit } from 'lodash'
import VueNotifications from 'vue-notifications'
```

因为项目中默认有 lodash，其实和 ramda 差不多，只不过 lodash 的参数传递顺序跟 ramda 相反。假如使用的是函数式编程，则先传递回调的 ramda 会更符合函数式编程规范。

`fn = debounce(innerFn, 300)` 是函数去抖动，就是让 innerFn 函数永远延迟 300 毫秒执行，当 300 毫秒内多次执行 fn 时，会从最后一次执行 fn 的时间开始计时，超过 300 毫秒才会真正执行 innerFn。这样的好处就是当 innerFn 函数被频繁触发的时候，只有最后一次调用超过 300 毫秒之后才真正执行，这样可以提高效率。

- 确认每一次字段的配置项

```
/** need config sample
 * eslint-disable */
const configSample = [
  {
    push_name: 'username',
    show_name: '用户名',
```



```

    defaultValue: '',
    type: 'text'
  }
]
/* eslint-enable */

```

我们期望得到这样的配置项，`push_name` 是提交表单的字段，`show_name` 是显示的名字，这可以从之前的 `map` 中获取，`defaultValue` 是默认值，`type` 是 `input` 的类型。

• 构建 v-form

我们直接使用 `Vue.component` 来构建组件，第二个参数是配置项，这里不写 `template`，而是写 `render` 函数，由于项目默认配置了 `vue-app` 插件，所以自带了转移 `jsx` 的功能。但要注意的是，`jsx` 语法只能写到 `render` 函数里，否则不会自动注入依赖。

首先定义 `props` 属性，`URL` 是我们后台提交的地址，`config` 则是前面我们约定好的配置项数组。然后在 `data` 里初始化数据，即 `push_name` 作为 `key`、`defaultValue` 作为 `value` 放入 `data` 中。

`notifications` 里是插件通知，其中的 `message` 是可以被覆盖的。`methods` 里的 `change` 用于绑定表单 `onKeyUp` 事件，但是由于 `jsx` 也支持 `v-model`，所以要有限使用 `v-model`。

`push` 则是推送数据到后端，通过 `e.preventDefault()` 阻止浏览器的默认行为，比如跳转页面等。`lodash` 的 `map` 有一些默认行为，当我们直接传入 `push_name`，它会获取其中对象的 `push_name` 的值。得到的值类似于 `['id', 'code', 'createdAt']` 这样的数据。推送完成之后会通过 `emit` 触发父组件的 `finish` 事件，在 `finish` 事件里隐藏 `model`。

这里笔者没有把构建表单的 `config.map` 拿出来，是因为之前所提到的注入依赖的问题，即 `h` 函数，大家可以尝试一下，把它写到 `render` 外面，然后打印它，看它有什么不同。

```

const VForm = Vue.component('v-form', {
  props: {
    model: {
      type: String,
      required: true
    },
    url: {
      type: String,
      required: true
    },
    config: {

```

```

    type: Array,
    required: true
  },
},
data () {
  return {
    /** init the config { push_name -> defaultValue }
    ...this.config.reduce((acc, { push_name, defaultValue }) => {
      acc[push_name] = defaultValue || ''
      return acc
    }, {})
  }, {}
},
notifications: {
  showSuccessMsg: {
    type: VueNotifications.types.success,
    title: '操作成功',
    message: '数据创建完成!'
  },
  showErrorMsg: {
    type: VueNotifications.types.error,
    title: '操作失败',
    message: '空'
  }
},
methods: {
  /** onKeyUp bind fn, use v-model replace it.
  change (event) {
    ~debounce(() => {
      this.$set(this, event.target.name, event.target.value)
    }, 520)()
  },
  async push (e) {
    e.preventDefault()
    const data = map(this.config, 'push_name')
    try {
      const { data: model } = await this.$api.create(this.model, pick(this, data))
      if (model) {

```

```

        this.showSuccessMsg()
        omit(this, data)
        this.$emit('finish')
    }
  } catch (e) {
    this.showErrorMsg({
      message: e.response.data
    })
  }
},
render () {
  /** build forms
  const forms = this.config.map(({ push_name, type, defaultValue,
  show_name }) => {
    switch (type) {
      case 'textarea':
        return (
          <div class="field">
            <label class="label">{show_name}</label>
            <div class="control">
              <textarea class="textarea" name={push_name} v-model={this
[push_name]}>
                {defaultValue}
              </textarea>
            </div>
          </div>
        )
      default:
        return (
          <div class="field">
            <label class="label">{show_name}</label>
            <div class="control">
              <input
                class="input"
                domPropsType={type}
                name={push_name}
                v-model={this[push_name]}

```

```

        placeholder={show_name}
      />
    </div>
  </div>
)
}
})
/** render jsx
return (
  <div class="form">
    <h2>{this.model}</h2>
    {this.data}
    <form>
      {forms}
      <div class="field ">
        <div class="control">
          <button class="button is-primary" onClick={this.push}>
            提交数据
          </button>
        </div>
      </div>
    </form>
  </div>
)
}
})

```

```
export default VForm
```

• 修改 ModelList.vue

引入依赖：

```

import VForm from '../components/VForm'
import { mapObjToArray } from '../utils'

```

• 注册组件与添加数据

添加注册组件的逻辑配置、控制表单显示的状态变量，以及得到状态，这些都是添加的方法与变量。


```
export default {
  data: {
    open: false
  },
  components: {
    VForm
  },
  computed: {
    ...mapState(['tables'])
  }
}
```

- 添加方法

nomarlizeCreateConfig 用来将原来 tables 里的对象格式化为我们之前约定的 config 格式。

```
methods: {
  hideModal() {
    this.open = false
  },
  finish() {
    this.open = false;
    this.loadItems()
  },
  nomarlizeCreateConfig(config) {
    return mapObjToArray(
      (val, key) => ({
        show_name: this.$dbMap[key] || key,
        push_name: key,
        defaultValue: '',
        type: val
      })),
    config
  )
}
```

- 添加样式

修改鼠标指针:

```
.create {  
  cursor: pointer;  
}
```

- 添加模板

添加到合适的位置:

```
<div class="column">  
  <h5 class="title is-6">基本详情</h5>  
  <ul>  
    <li>最大页数 {{ maxPage }}</li>  
    <li>当前页面 {{ page }}</li>  
    <li>当前分页 {{ limit }}</li>  
    <li class="create" @click="open=true">创建</li>  
  </ul>  
</div>  
  
<vue-modaltor :visible="open" @hide="hideModal">  
  
<v-form :model="type" :config="nomarlizeCreateConfig(tables[type])" :url="'/  
api/model/' + type" @finish="finish" />  
</vue-modaltor>
```

vue-modaltor 是插件提供的全局组件,用 visible 来控制显示,而 hide 事件用来触发关闭静态框,然后显示 v-form 组件。

6. 测试

- 单击“创建”按钮创建测试
- 当我们提交数据有问题时会显示错误信息。
- 成功后会提示“操作成功,数据创键完成!”

4.4 修改逻辑

之前写好并显示的 JSON 视图还不是特别友好，现在我们来将其做成表格的形式，并添加删除和修改功能。

1. 添加图标

在 build/index.html 的 head 区域添加图标样式。其中 integrity 是一串校验码，而 crossorigin 跨域是任意域。样式都是由 fontawesome 提供的。

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.0.8/css/solid.css" integrity="sha384-v2Tw72dyUXeU3y4aM2Y0tBJQkGfplR39mxZq1TBDUZA9BGoC40+rdFCG0m101Xk" crossorigin="anonymous">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.0.8/css/fontawesome.css" integrity="sha384-q3jl8XQulOpdLgGFvNRnPdJ5VilCvgsDQTB6owSOHWLAurxul7f+JpUOVdAiJ5P" crossorigin="anonymous">
```

2. 添加 API 接口

修改 client/api.js，添加更新和删除方法：

```
export function update (model, id, data) {
  return fetch({
    method: 'PUT',
    url: `/${model}/${id}`,
    data
  })
}
```

```
export function del (model, id, data) {
  return fetch({
    method: 'DELETE',
    url: `/${model}/${id}`,
    data
  })
}
```

3. 修改视图

将视图修改为 table 视图，从 tables 里获取字段名字，渲染表头，然后从 displayItem 里渲染数据，最后一行为操作栏，里面的图标都是由之前我们添加的图标样式文件提供的。

```
<!-- <div class="json" v-html="pretty(displayedItems)"></div> -->
<div class="container">
  <table class="table table__model-list">
    <thead>
      <th v-for="(_, val) in tables[type]" :key="val" v-text="val"></th>
      <th>操作</th>
    </thead>
    <tr v-for="item in displayedItems" :key="item.id">
      <td :key="text" v-for="text in item" v-text="text"></td>
      <td><i class="fas fa-trash-alt" @click="remove(item.id)"></i><i
class="fas fa-edit" @click="edit(item)"></i></td>
    </tr>
  </table>
</div>
```

添加数据与方法，editItem 表示正在编辑的数据，也增加了 notifications 的提示消息。edit 方法是用来显示修改表单的方法，finish 则是操作完成的时候触发的方法，它会重置正在编辑的项目，并重新载入数据，remove 则是直接提交删除请求，成功后显示消息。

那么如何区分是编辑项目还是新建项目呢？通过 editItem 区分即可，不过在修改完成之后要清除 editItem，否则还是编辑项目。

```
data() {
  return {
    editItem: null
  },
  computed: {
    ...mapState(['tables'])
  },
  notifications: {
    showSuccessMsg: {
      type: VueNotifications.types.success,
      title: '操作成功',
```



```
      message: '完成!'
    },
    showErrorMsg: {
      type: VueNotifications.types.error,
      title: '操作失败',
      message: '空'
    }
  },
  methods: {
    hideModal() {
      this.open = false
    },
    edit(item) {
      this.editItem = item
      console.log(this.editItem)
      this.open = true
    },
    finish() {
      this.open = false
      this.showSuccessMsg()
      this.editItem = null
      this.loadItems()
    },
    async remove(id) {
      try {
        const data = await this.$api.del(this.type, id)
        if (data.status === 200) {
          this.showSuccessMsg({ message: '删除成功' })
          this.loadItems()
        }
      } catch (e) {
        this.showErrorMsg({ message: e.response.data })
      }
    },
  }
}
```

有的字段可能太长了，所以显示就不太友好，直接让它横向滚动即可，按住 Shift 键滚动

鼠标就是横向滚动。

```
.container {
  overflow: scroll;
}
.table.table__model-list {
  min-width: 200%;
}
```

4. 修改静态框逻辑

修改 components/VForm.js 文件。

添加一个 edit 属性,通过 config 计算出 fieldsNames 属性,它是提交的对象 key 的数组,传递给 pick 方法使用。监听了 edit 属性,当变化的时候,复制到自身属性上。handleRowData 用于获取对象上所有要提交的数据。pushMethod 根据是否有 edit 属性来确认是更新还是创建方法。resets 是重置属性的方法,push 则是统一推送数据的方法,即在视图上被绑定的方法。

```
{
  props: {
    edit: { type: null }
  },
  computed: {
    fieldsNames() {
      return map(this.config, 'push_name')
    }
  },
  watch: {
    edit(newVal) {
      Object.assign(this, newVal)
    }
  },
  methods: {
    handleRowData() {
      return pick(this, this.fieldsNames)
    },
    pushMethod(model, data) {
      return this.edit ? this.$api.update(model, data.id, data) : this.$api.create(model, data)
    }
  }
}
```

```
    },
    resets() {
      this.fieldsNames.map(v => {
        this[v] = ''
      })
    },
    async push(e) {
      e.preventDefault()
      try {
        const push_row_data = this.handleRowData()
        const { data: model } = await this.pushMethod(this.model,
push_row_data)
        if (model) {
          this.resets()
          this.$emit('finish')
        }
      } catch (e) {
        this.showErrorMsg({
          message: e.response.data
        })
      }
    },
  },
}
```

5. 测试

修改

单击“编辑”按钮，编辑数据。

修改数据，单击“提交数据”按钮，如图 4-2 所示。

Tag

ID 号

1

name

This a tag

created_at

2018-03-19T00:00:00.000Z

updated_at

2018-03-19T06:31:27.000Z

提交数据

图 4-2

结果如图 4-3 所示。



图 4-3

```
2018-03-24 10:57:10.510 INFO 3600 [model] UPDATE `Tags` SET `name`='This a tag3', `updated_at`='2018-03-24 02:57:10' WHERE `id` = 1 (1ms)
```

后台也显示了修改的 SQL 语句。

删除

单击“删除”按钮，如图 4-4 所示。

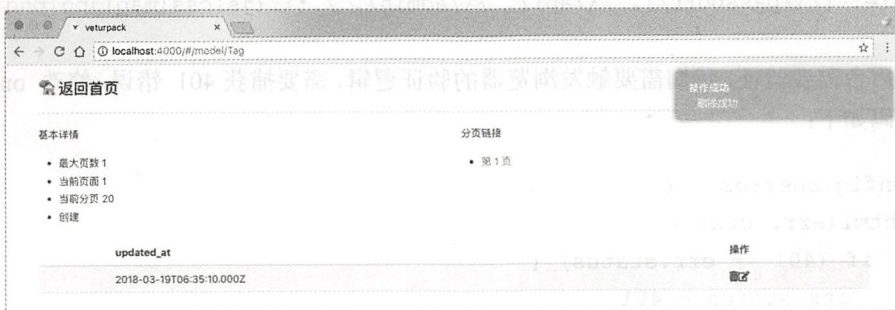


图 4-4


```
[model] DELETE FROM `Tags` WHERE `id` = 1 LIMIT 1 (1ms)
```

后台成功地执行了 SQL，前台的数据也直接刷新了。

6. 构建

开始构建

修改 build/config.js 中的 publicPath 为 '/public'。

```
npm run build
```

假如有一个关于 UglifyJs 的错误，则不用管它，可以正常工作。

构建完成后，把生成的 dist 目录下的所有文件复制到后端的 app/public 目录下。然后把 index.html 修改为 admin.html，完成后把它移动到 views 目录下。

配置后台

添加路由：

```
router.get(
  '/admin',
  auth({
    name: 'dd',
    pass: '888888'
  }),
  ctx => ctx.render('admin')
)
```

这时会报一大堆 JWT 错误，所以我们要修改一下 JWT 的正则。修改 ignore 为下面的内容。

```
ignore: [/\/passport/i, /\/api/, /\/admin/, /\.*\.(js|css|map|jpg|png|ico)/]
```

因为后台需要验证，我们需要触发浏览器的验证逻辑，需要捕获 401 错误。修改 onerror 的配置，代码如下：

```
config.onerror = {
  html(err, ctx) {
    if (401 == err.status) {
      ctx.status = 401
      ctx.set('WWW-Authenticate', 'Basic')
```

```
ctx.body = 'input your use info'
return
}
```

输入用户名与密码之后就可以正常地访问了。

7. 小结

现在后台的逻辑就告一段落了，先保证能用，尽管显得比较简陋，再考虑扩展。本章我们大概用了 600 多行代码实现了一个自动化的后台，其实还有很多的抽象空间，完全可以像 Python 的 Django 那样实现更多功能，动手能力强的读者可以在这个基础上扩展更多的功能。

5 chapter

第 5 章 前端界面设计与实现

5.1 搭建前端开发环境

为了获取更好更丰富的内容，前端使用 TypeScript 与 Vue SSR 来处理 SEO 问题。使用 TypeScript 最重要的就是书写定义文件，这样才能规避错误，本章我们将一一解决遇到的问题。

5.1.1 开始

搭建基本的开发栈其实非常费时间，而且基础不好的读者还不一定能搭建成功，笔者自己也搭建了很久，特别是在 Webpack4 与 Webpack3 过渡的阶段，其中有不少插件没有做兼容，一个大的版本升级就修改了不少 API。目前 Vue 官方脚手架里都是 Webpack3 版本的，而且脚手架里不提供 SSR 和 TypeScript 的基本环境。

```
git clone https://github.com/MiYogurt/vue-typescript-starter.git
```

首先将 server.ts 里的端口改成 4000，与之前的后端端口一样，以避免跨域限制。

笔者一直维护更新它，读者假如想要保持一致，可以回退到 44ffce 版本号，即第三次提交。

1. 安装依赖

```
cd vue-typescript-starter
npm install
npm run dev
```

在客户端会自动打开 8888 端口，它是用来分析打包依赖的工具。上面的 bundle.js 部分是之前打包的，而下面的 update.js 是更新的部分，右边则是 Vue 组件，如图 5-1 所示。

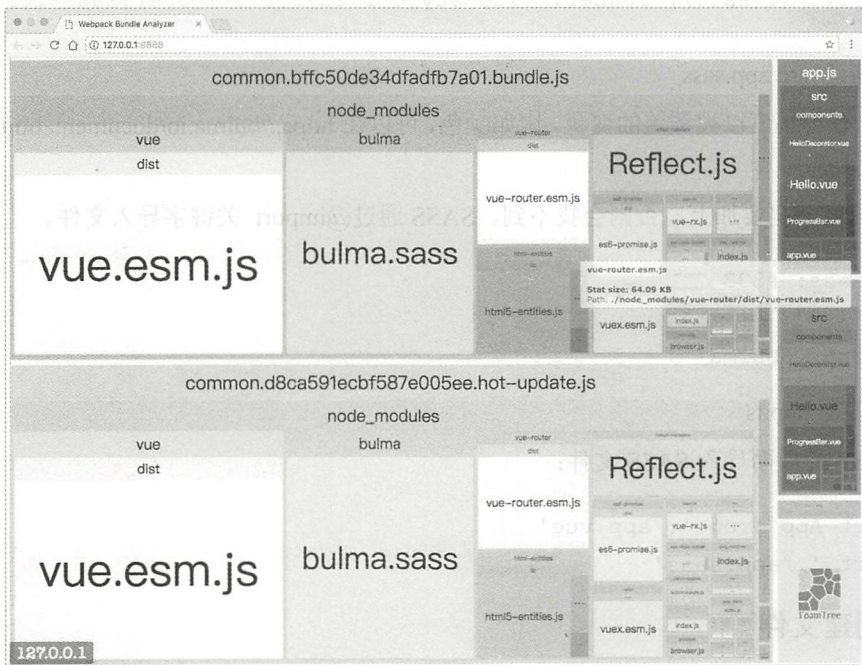


图 5-1

2. 删除缓存

由于之前 4000 端口注册了 serviceWork，所以现在访问可能看到的是原始的后台页面，打开开发者工具，进入 Application 选项卡，单击 unregister 链接取消注册，然后刷新页面即可。

5.1.2 创建 Header 头部

1. 安装样式依赖

bulma 是用 SASS 写的一个 CSS 库，我们使用 bulma 来写样式文件，笔者比较常用的工

具还是 stylus，而考虑到 SASS 更加流行，所以以 SASS 为例来写样式，两者其实差不多，只不过 stylus 更接近 JavaScript 而已。

```
npm i bulma -S
npm i sass-loader node-sass -D
```

添加 webpack 的配置，修改 configs/webpack/base.ts，把下面的配置添加到 rules 配置项中。

```
{ test: /\.sass$/, loader: 'vue-style-loader!css-loader!sass-loader' },
```

- 新建 src/app.sass

bulma 有一些可以被覆盖的变量，比如颜色，可以在 <https://bulma.io/documentation/overview/colors/> 上找到。

~ 波浪号一定要加上，否则会找不到。SASS 通过 @import 关键字导入文件。

```
$success: #333
```

```
@import '~bulma'
```

- 修改 app.ts

在 App 的后面导入 SASS 文件：

```
import App from './app.vue'
import './app.sass'
```

2. 创建文件

新建 src/components/layouts/AppHeader.vue。

bulma 的 NavBar 组件完全符合我们的要求，相关文档在这里：<https://bulma.io/documentation/components/navbar/>。然后我们使用 pug 模板把 HTML 转换为 pug 格式。这个网站 (<http://html2jade.vida.io/>) 可以帮助我们做这件事。jade 是 pug 的老名字。

pug 可以参考 <https://pugjs.org/api/getting-started.html>，也可以观看笔者录制的视频教程，地址为 <https://nodelover.me/course/pug>。

假如感觉 pug 太难，直接写 HTML 也没问题，这只是一个团队的标准问题而已。这里为什么要使用 AppHeader 作为名字呢？因为 header 与 HTML 标签有冲突，假如你使用过 element-ui，你可能会发现它们都有一个 el 的命名空间，而我们这里全局都是以 App 为命名空间的，约定的规则可以参考官方的文档 (<https://cn.vuejs.org/v2/style-guide>)。

```

<template lang="pug">
nav.navbar: .container
  .navbar-brand
    router-link.navbar-item(to="/" exact-active-class="no-active")
      img(src="/public/logo.png", alt="喵喵看世界")
  .navbar-burger.burger(data-target="navbar-app")
    span
    span
    span
  #navbar-app.navbar-menu
    .navbar-start
      router-link.navbar-item(to="/team") 团队
      router-link.navbar-item(to="/blog") 博客
      router-link.navbar-item(to="/podcast") 播客
    .navbar-end
      router-link.navbar-item(to="/signup") 注册
      router-link.navbar-item(to="/signin") 登录
</template>

<script lang="ts">
import { Vue, Component } from 'vue-property-decorator';
@Component({
  name: "AppHeader"
})
export default class extends Vue {}
</script>

<style lang="sass" scoped>
.router-link-exact-active
  color: #000
  background: whitesmoke
</style>

```

笔者基于模板做了一些修改, 由于 Vue 的路由是通过 router-link 跳转的, 所以我们要用 router-link 进行跳转, router-link 会自动匹配路由, 加上对应的 class 比如 router-link-exact-active、router-link-active, 让我们可以识别出来。

是否带 exact 的区别就是, exact 表示严格匹配, 即 /static 路径, 在严格匹配模式之后,

只会匹配路径为 /static 的 router-link，而非严格模式，还会匹配它的父路径，比如 / 路径。对于 logo 我们不希望它有什么样式，所以我们通过 exact-active-class 随便定义一个 class 名字，用以覆盖 router-link-exact-active。

我们通过 vue-property-decorator 所提供的装饰器来写 class 风格的 Vue 组件，所有默认的属性都可以写到装饰器的配置项中，比如 name，一定要给定这个 name，否则它会变成匿名组件，即使给出 class 名字，Vue 也不会知道组件的名字。

3. 添加移动端效果

- 修改模板

当单击 navbar-burger 选项的时候，就会给它们加上 is-active class。读者朋友可能会注意到，笔者给 :class 加上了引号，这是因为 “:” 在 JS 中代表属性赋值，容易产生冲突。还有好几种解决办法，我们这里制定一个标准，即不用缩写。这里只是做演示，笔者会按照标准来修改代码。

```
nav.navbar: .container
  .navbar-brand
    router-link.navbar-item(to="/" exact-active-class="no-active")
      img(src="/public/logo.png", alt="喵喵看世界")
    .navbar-burger.burger(':class'="{ 'is-active': isActive }" data-target=
"navbar-app" @click="toggleNavBar")
      span
      span
      span
  #navbar-app.navbar-menu(':class'="{ 'is-active': isActive }")
    .navbar-start
      router-link.navbar-item(to="/team") 团队
      router-link.navbar-item(to="/blog") 博客
      router-link.navbar-item(to="/podcast") 播客
    .navbar-end
      router-link.navbar-item(to="/signup") 注册
      router-link.navbar-item(to="/signin") 登录
```

- 修改 class

属性前面的就是 data 状态，而方法就是 Vue 里的 methods。

```
export default class extends Vue {
```

```

public isActive: boolean = false

public toggleNavBar(){
  this.isActive = !this.isActive
}
}

```

现在在移动端就可以像这样打开菜单栏了，如图 5-2 所示。



图 5-2

4. 实现生命周期函数的钩子

有的时候我们可能要实现生命周期函数的钩子，输入 `created` 会发现没有提示，生命周期函数在 `ComponentOptions` 接口上，我们实现这个接口就好了。

像下面这样实现 `ComponentOptions` 接口，但是必须要实现一个方法，否则会报错，而对于 `watch` 的写法可以通过装饰器去写，也可以在配置对象里写，更多的内容可以参考 `vue-property-decorator` 库。

```

import { Vue, Component, Watch } from 'vue-property-decorator';
import { ComponentOptions } from 'Vue';
@Component({
  name: "AppHeader",
  // watch: {
  //   isActive(oldVal, newVal) {
  //     console.log(oldVal + ' -> ' + newVal);
  //   }
  // }
})
export default class extends Vue implements ComponentOptions<Vue> {
  public isActive: boolean = false

  public toggleNavBar(){

```



```

    this.isActive = !this.isActive
  }
  @Watch('isActive')
  isActiveC(oldVal, newVal) {
    console.log(oldVal + ' -> ' + newVal);
  }
  created() {
    console.log("app-header created")
  }
}

```

5.1.3 将变量提取出来

新建 src/_vars.sass:

```
$link: #0a0a0a
```

修改 src/app.sass:

```

@import 'vars'
@import '~bulma'

```

body

```

  background: #f6faff
  min-height: 100vh

```

安装依赖:

```
npm install sass-resources-loader -D
```

修改 configs/webpack/client 配置文件:

```

{
  test: /\.vue$/,
  loader: 'vue-loader',
  options: {
    extractCSS: isProd,
  }
}

```

```
    loaders: {
      scss: 'vue-style-loader!css-loader!sass-loader',
      sass: 'vue-style-loader!css-loader!sass-loader?indentedSyntax!sass-resources-loader?resources=' + getPath('src/_vars.sass'),
      styl: 'vue-style-loader!css-loader!stylus-loader'
    }
  },
  { test: /\.sass$/, loader: 'vue-style-loader!css-loader!sass-loader!sass-resources-loader?resources=' + getPath('src/_vars.sass') },
```

修改 Vue 和 SASS，处理链，通过 query 传递确实有一些不合适，还可以优化一下。即将 vue-loader 里的 SASS 处理改成如下形式：

```
sass: [
  'vue-style-loader',
  'css-loader',
  {
    loader: 'sass-loader',
    options: {
      indentedSyntax: true
    }
  },
  {
    loader: 'sass-resources-loader',
    options: {
      resources: getPath('src/_vars.sass')
    }
  }
],
```

5.1.4 添加路径重写

```
npm i tsconfig-paths-webpack-plugin -D
```

- 修改 webpack/config/base.ts

```
resolve: {
```

```

alias: {
  public: getPath('public'),
  vue$: 'vue/dist/vue.esm.js'
},
extensions: ['.ts', '.tsx', '.js', '.vue', '.json'],
plugins: [
  new TsconfigPathsPlugin({
    configFile: getPath('.'),
    logLevel: "info",
    extensions: [".ts", ".tsx", "vue", "sass"]
  })
],

```

- 修改项目目录下的 `tsconfig.json`，添加新选项，不要删除原来的选项

```

{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@C/*": ["src/components/*"]
    }
  },
}

```

- 修改 `src/tsconfig.json`

```

{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@C/*": ["src/components/*"]
    }
  },
}

```

- 重启 VSCode

这样会重启 `tsserver`，之所以有两个文件，是因为笔者给 `src` 下配置的是 `esnext` 的模块，

便于 tree-shaking, 但是由于 ES6 模块即以上 TypeScript 会报错说不支持动态导入, 所以此处写了一个兼容的 `_import` 方法来导入, 并禁用了警告, 如图 5-3 所示。

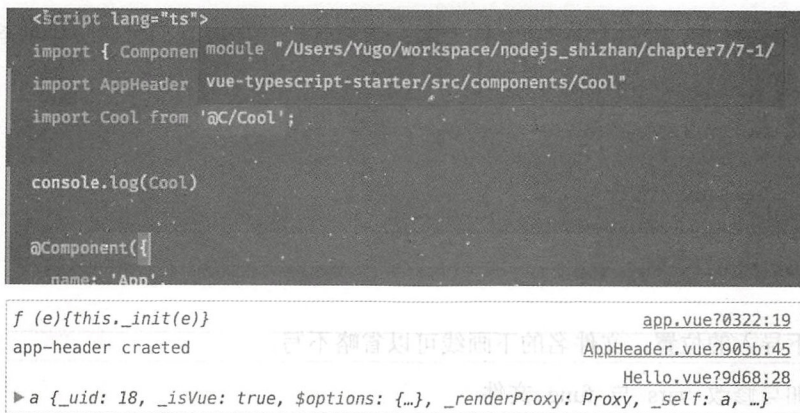


图 5-3

浏览器输出了 Cool。

5.2 AppFooter 组件

写组件之前, 先把 footer 里的 log 给导出来, 注意缩放大小一定要合适, 可以选用 @2x 的大小, 如图 5-4 所示。

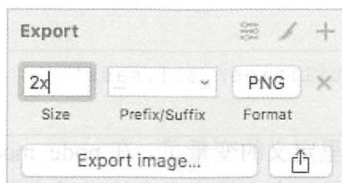


图 5-4

5.2.1 做一些配置

目前虽然可以重写 Bulma 中的变量, 但是在 Vue 组件里想要使用 Bulma 的变量该怎么办? 假如可以使用这些变量, 则可以非常容易地保持 UI 的一致性。

- 修改 config/webpack/client.ts

```
resources: [ getPath('src/sass/_vars.sass'), getPath('src/sass/_func.sass') ]
{ test: /\.sass$/, loader: 'vue-style-loader!css-loader!sass-loader!sass-
```



```
resources-loader?resources=' + getPath('src/sass/_vars.sass') + '&resources='
+ getPath('src/sass/_func.sass') },
```

修改以上两处，因为我们的样式文件太多，需要管理，所以我们把它集中放到 `src/sass` 文件夹里。

- 修改 `src/app.sass`

```
@import './sass/vars'
@import './sass/func'
@import '~bulma'
```

修改一下导入的位置，文件名的下画线可以省略不写。

- 添加与修改 `vars` 与 `func` 文件

`src/sass/_func.sass:`

```
=no-padding-lr
padding-left: 0 !important
padding-right: 0 !important
```

`src/sass/_vars.sass:`

```
$link: #0a0a0a

@import '~bulma/sass/utilities/_all.sass'
```

这样我们就可以使用 `bulma` 里定义的变量了。在 `node_modules` 的 `bulma/sass/utilities/initial-variables.sass` 里可以找到这些变量，在文档中也会给出对应的修改变量。

这些变量定义的语法都是 `!default`，表示假如变量未定义则使用该值，所以这些变量可以被覆盖。

5.2.2 创建 `src/components/layouts/AppFooter.vue`

1. 添加图标文件

在 `src/index.template.html` 里添加图标的 CSS 文件。

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.0.9/
```

```
css/all.css" integrity="sha384-5SOiIsAziJl6AWe0HWRKTXlfcSHKmYV4RBF18PPJl73Kz  
n7jzMyFuTtk8JA7QQG1"  
crossorigin="anonymous">
```

2. 第一列

- script

可以写全一点，像下面这样：

```
import { Vue, Component } from 'vue-property-decorator'  
@Component({  
  name: 'AppFooter'  
})  
export default class extends Vue {}
```

也可以像 JS 那样，直接导出一个对象字变量。

```
export default {  
  name: 'AppFooter'  
}
```

- pug

```
footer.footer: .container  
  .columns.is-multiline  
    .column.is-3#logo-panel  
      .tile.is-ancestor  
        .tile.is-4  
          img.footer-logo(src="/public/footer-logo.png")  
        .tile.is-8.is-hidden-touch.is-vertical  
          h3.title.is-3.has-text-weight-normal 猫览  
          p.subtitle.has-text-grey-light.has-text-weight-normal 向世界展示你的才华  
          p.is-size-6.has-text-grey-light.has-text-weight-normal 猫览，每一位设计  
          猫的温馨家园，在这里分享、传递设计  
          ul.social-link  
            li: a(href="javascript:void(0);"): span.icon.is-medium: i.fab.fa-  
dribbble.fa-lg  
            li: a(href="javascript:void(0);"): span.icon.is-medium: i.fab.fa-  
twitter.fa-lg
```



```

li: a(href="javascript:void(0);"): span.icon.is-medium: i.fab.fa-
facebook.fa-lg
li: a(href="javascript:void(0);"): span.icon.is-medium: i.fab.fa-
instagram.fa-lg
li: a(href="javascript:void(0);"): span.icon.is-medium: i.fas.fa-
globe.fa-lg

```

假如只有一个元素，则使用冒号“:”可以缩短代码，表示如下：

```

.footer
  .container
    .columns
      p 我是内容
.footer: .container: .columns
  p 我是内容

```

`.columns` 与 `column` 是网格布局的一种，它支持英文的几分之几这种语法，对于这种语法可能对外文比较友好，但是对中文却不太友好。它支持 12 栅格布局，意思就是一行分为 12 等分，通过加 `is-xx` 类名来指定所占的宽度，比如 `is-3`、`is-4`、`is-12` 等。

`.is-multiline` 表示可以多行显示，`columns` 的本质还是 `flex` 布局。而 `tile` 同 `columns` 比较类似，不过它不支持几分之几的语法。

`tile` 的源码如下，其实它非常简单。`is-ancestor` 通过负值外边距减小边距。`is-parent` 会添加内边距，用以抵消负值外边距，`is-child` 会清除负外边距，`is-vertical` 会纵向排列。在桌面端浏览器中，非 `is-child` 都是 `flex` 布局。

```

.tile
  align-items: stretch
  display: block
  flex-basis: 0
  flex-grow: 1
  flex-shrink: 1
  min-height: min-content
  &.is-ancestor
    margin-left: -0.75rem
    margin-right: -0.75rem
    margin-top: -0.75rem
    &:last-child

```



```

margin-bottom: -0.75rem
&:not(:last-child)
margin-bottom: 0.75rem
&.is-child
margin: 0 !important
&.is-parent
padding: 0.75rem
&.is-vertical
flex-direction: column
& > .tile.is-child:not(:last-child)
margin-bottom: 1.5rem !important
+tablet
&:not(.is-child)
display: flex
@for $i from 1 through 12
  &.is-#{ $i }
    flex: none
    width: ( $i / 12 ) * 100%

```

那么 `min-height: min-content` 又是什么呢？其实就是它字面上的意思，元素的最小高度，至少要能容纳里面的内容。

图 5-5 所示是正常的布局。

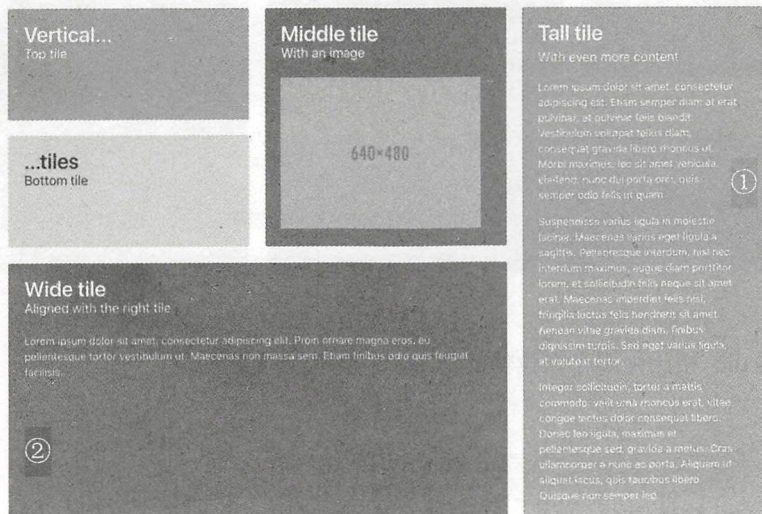


图 5-5



当我们给①部分的内容设置高度为 300px 时②部分的高度变小了，变成能容纳内容的最小高度。而①的部分，虽然内容超出了，但是由于 `align-items: stretch` 的拉伸，导致①部分与②部分保持在同一个水平线上，如图 5-6 所示。

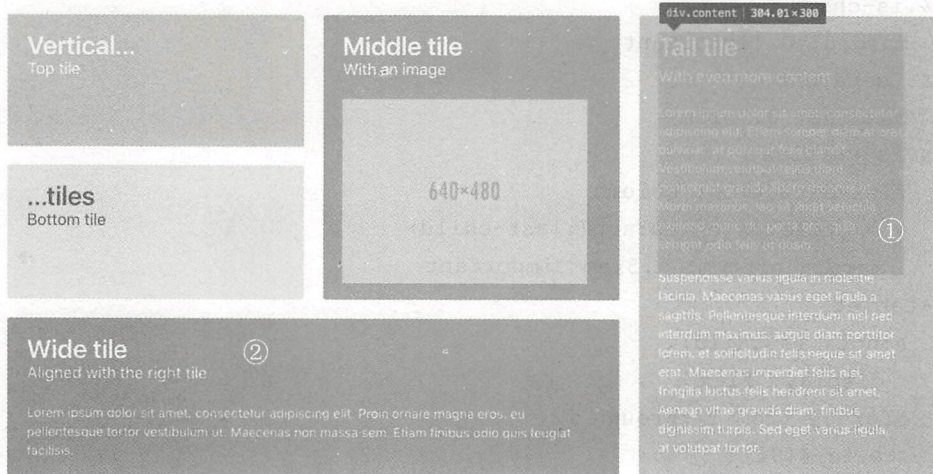


图 5-6

当我们去掉 `min-height: min-content` 属性后，它变成了图 5-7 这样，为了保持①部分与②部分的平行，内部盒子收缩都变形了。这里①部分的高度为 340，是因为有 20 的内边距，如图 5-7 所示。

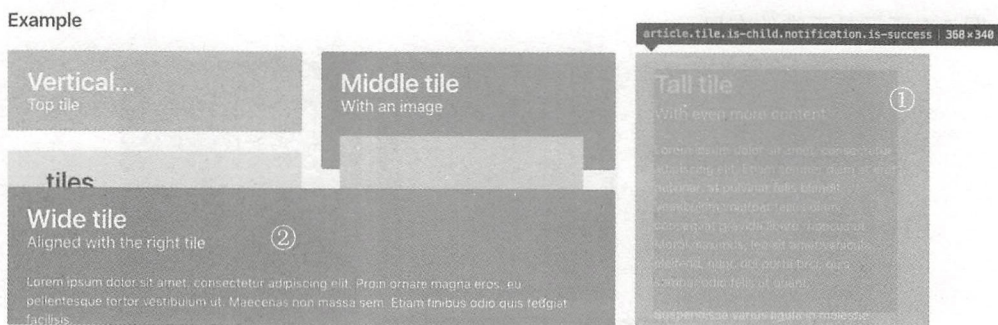


图 5-7

`columns` 同 `tile` 相比则不存在拉伸的情况，它的源码大致如下：



```

1  $column-gap: 0.75rem !default
2
3  * .column...
4
5  .columns
6
7      margin-left: (-$column-gap)
8      margin-right: (-$column-gap)
9      margin-top: (-$column-gap)
10
11  * .last-child...
12  * .not(:last-child)...
13
14  // Modifiers
15
16  * .is-centered...
17  * .is-gapless...
18  * .is-mobile...
19  * .is-multiline...
20  * .is-vcentered...
21
22  // Responsiveness
23
24  * .tablet...
25  * .desktop...
26
27  // Modifiers
28
29  * .is-desktop...
30
31  * @if $variable-columns...
32
33

```

• style

添加一些样式，object-fit: contain 是为了保持原始缩放。

```

.footer
    background: transparent
.title.is-3 ~ .subtitle
    margin-top: -1rem
.footer-logo
    object-fit: contain
    width: 100%
    height: 73px
.social-link
    padding: .7em 0
li
    display: inline-block
a

```



```

    color: #e6e6e6
    &:hover
      color: $link
#logo-panel
  margin-bottom: 0
  padding-bottom: 0

```

3. 第二列

注意缩进，`column` 应该保持在同一个级别，`//-` 则表示注释，这里使用了两种方式，一种是使用 `tile`，另一种是自己写，其实有的时候过多地使用框架中的修饰 `class`，会导致类名很长，请读者自行斟酌，“鱼和熊掌不可兼得”。

- pug

```

.column.is-3.more-link
  .tile.is-ancestor: .tile.is-vertical
    .tile.is-parent.no-padding-lr
      .tile.is-child.is-4
        h3.title.is-6.is-marginless 更多链接
      .tile.is-child.is-hidden-touch
        h3.title.is-6.is-marginless 关于
    .tile
      //- ul.tile.is-parent.no-padding-lr.flex-wrap
      //- li.tile.is-4: a.is-size-7.has-text-grey-light(href="#") UI 中国
      //- li.tile.is-4: a.is-size-7.has-text-grey-light(href="#") 联系我们
      //- li.tile.is-4: a.is-size-7.has-text-grey-light(href="#") 意见反馈
      //- li.tile.is-4: a.is-size-7.has-text-grey-light(href="#") 用户反馈
      ul.tile.is-flex.link
        li: a.is-size-6(href="#") UI 中国
        li: a.is-size-6(href="#") 联系我们
        li: a.is-size-6(href="#") 意见反馈
        li: a.is-size-6(href="#") 用户反馈
        li: a.is-size-6(href="#") 版权声明
        li: a.is-size-6(href="#") 用户信息

```

- style

`+mobile` 表示媒体查询，它是一个 `mixin` 混合。写在它下面的部分，表示在移动端呈现的样式。



```
.more-link
+mobile
  .tile.is-ancestor
    margin: 0
  .link
    width: 100%
    flex-wrap: wrap
  li
    width: 33%
    padding: .4em 0
    a
      color: $grey
      &:hover
        color: $link
```

这是 mobile 的 mixin 的源码，mixin 以 “=” 开头，引用的时候用 “+”，在 SCSS 文件中有另外的语法，这里使用这种方式完全够了。

```
=mobile
  @media screen and (max-width: $tablet - 1px)
    @content
```

这里的 @content 其实就表示上面的 .tile.is-ancestor 中的内容。

```
=border-radius($radius)
  border-radius: $radius

.box
  +border-radius(10px)
```

当然，也可以把 mixin 当作这样的函数来使用，尽管函数这个词不是特别正确，真正的函数应该用 @function 去定义。

例如：

```
@function findColorInvert($color)
  @if (colorLuminance($color) > 0.55)
    @return rgba(#000, 0.7)
```




```
@else
  @return #fff
```

colorLuminance 是另外一个函数，它判断明暗对比度，最后返回一个颜色。

4. 第三列

- pug


```
.column.is-3.campaign
  h1.title.is-6 活动
  img(src="http://via.placeholder.com/350x150")
  p.is-size-7 参加活动，获取新款 kindle
```

<http://via.placeholder.com> 是一个提供占位符图片的网站，这里我们输入的 URL 表示要求提供一个 350×150 的图片。

- style


```
.campaign
  padding-bottom: 0
  img
    width: 100%
  p
    padding-top: .7em
  .flex-wrap
    flex-wrap: wrap
  li
    padding: .1em 0
  p
    padding-top: .2em
```

5. 第四列

- pug


```
.column.is-3
  h3.title.is-6.is-marginless 统计
  .level
    .level-item.has-text-centered
      div
```





```
.heading 海报
.title 5,623,321
.level-item.has-text-centered
div
.heading 文章
.title 223,321
```

- style

```
.level
flex-direction: column
&-item
padding: 1.3em
+mobile
padding: 0.7em
```

5.2.3 网络识别信息

*pug

```
.column.is-12.is-size-7.has-text-grey-lighter#copyright
| 京 ICP 备 0789189798 号-1 / 京公网安备 318789791123 号 / Powered by 2008-2018
xxx.net
```

“|”表示新起一行写标签内的内容。

5.2.4 修改一下全局样式

```
.no-padding-lr
+no-padding-lr
```

```
body
background: #fff
min-height: 100vh
```





5.2.5 查看页面

图 5-8 所示是普通桌面浏览器的效果。



图 5-8

图 5-9 所示是移动端的效果。



图 5-9

5.2.6 提升编译速度

修改 configs/webpack/client.ts，使分析和提取的公共模块在开发时处于关闭状态。因为它们





都需要时间来分析模块,所以才会非常耗费性能,而 webpack 的热重载就已经能增量地更新了。

```
if (isProd) {
  clientConfiguration.plugins.push(new BundleAnalyzerPlugin())
  clientConfiguration.optimization = {
    splitChunks: {
      name: 'common',
      chunks: 'initial',
      minSize: 0
    },
    minimize: true
  }
}
```

还可以添加 Dll, Dll 其实就是动态链接库,就是把一部分不需要修改的库先打包出来,比如 vue.js、vue-router.js 等,它会生成一个 JSON 文件,webpack 读取这个 JSON 文件就会知道哪些打包过了。笔者在源码中做了测试,可以阅读源代码学习如何使用 Dll,而由于 Dll 会导致复杂度大幅上升,所以就不做更多讲解了。

5.3 首页

由于 Vue 文件使用的 TypeScript 提示是 Vetur 插件自己的服务,与默认的 TypeScript 的不同,所以在载入配置文件的时候会出现莫名其妙的情况,VSCode 会报错,但是会正常编译。忽略这些错误即可,假如实在感觉用 TypeScript 太难受了,则可以使用 tsc 编译出来,写 JS 文件,然后修改 webpack 的入口为 JS 即可。为了更好地排版与清晰地阅读,后面的章节笔者决定不再使用 pug 模板。

1. 配置

添加 CSS 和 JS 的处理,这样大家就可以编译 JS 文件,可以直接写 JS,因为插件里有 CSS,所以我们需要添加 CSS 文件处理的配置。

- 修改 configs/webpack/base.ts

```
{ test: /\.css$/, use: ['style-loader', 'css-loader'] },
{
  test: /\.jsx?$/,
  exclude: /node_modules/,
```





```
use: [{ loader: 'babel-loader', options: babelrc }],
},
```

- 修改 configs/webpack/client.ts

```
alias: {
  vue$: 'vue/dist/vue.esm.js',
  api: getPath('src/api/v1/client/index.ts')
}
```

添加 API 重写的路径。

- 修改 configs/webpack/server.ts

```
resolve: {
  alias: {
    api: getPath('src/api/v1/server/index.ts')
  },
  externals: [
    nodeExternals({
      whitelist: [/\.css$/, /\.styl$/, /\.vue$/]
    })
  ],
```

添加 API 重写的路径。

- 修改 src/shims/webpack-shims.d.ts

```
declare module 'api'
```

- 把全局的 SASS 文件放到 SASS 目录下

如图 5-10 所示，重新组织样式文件，记得添加全局的 vars 和 func，至于如何添加，请参考之前的 sass-resources-loader 配置，或者阅读源代码。

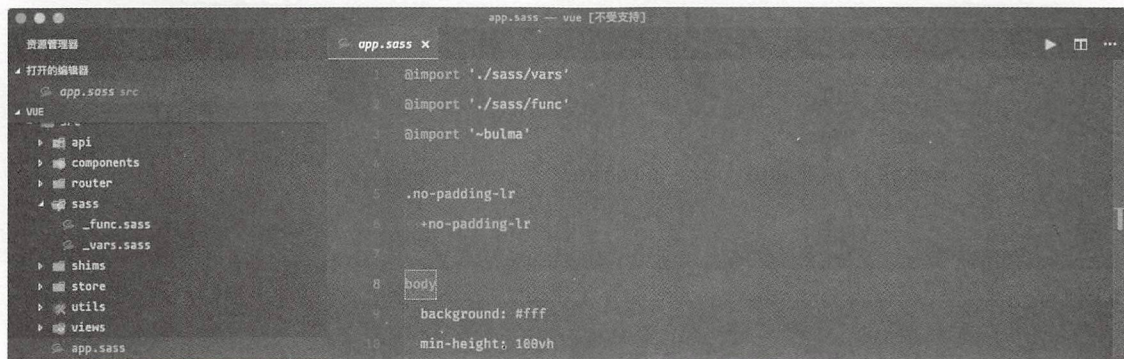


图 5-10

2. 安装依赖

现在我们需要两个插件，一个用于懒加载图片，另一个用于动画显示。

```
npm i vue-lazyload aos -S
```

3. 创建 src/views/index.vue

每次都要自己手动新建文件，有没有什么方法，可以像 Angular、Ruby On Rails 那样通过命令直接新建呢？可以自己写一个 cli 工具，但是这太麻烦了，还不如定义一个代码片段。于是 plop 登场了。

- 安装依赖

```
npm i plop -D
```

- 初始化配置文件

它会生成一个 plopfile.js，它与 gulp 一样支持多种格式，比如 .coffee、.ts、.babel.js 等，由于官方没有提供 d.ts 文件，所以基本用 TypeScript 写，也没什么提示。反而用 coffeescript 来写会更简单些，对于配置文件 coffee 格式是一个好的选择。

```
npx plop -init
```

修改内容如下：

```
module.exports = function(plop) {  
  plop.setGenerator('page', {  
    description: 'new vue page',
```



```
prompts: [
  {
    type: 'input',
    name: 'name',
    message: 'page name please'
  }
],
actions: [
  {
    type: 'add',
    path: 'src/views/{{name}}.vue',
    templateFile: 'templates/page.hbs'
  }
]
}))
}
```

page 是命令，prompts 是询问的问题，这里表示询问一个 name 的值，actions 是做的动作，这里的动作就是将 page.hbs 复制到 src/views 下，而 name 则是询问得到的变量值。

- 添加模板，新建 templates/page.hbs

```
<template>
  <div>
  </div>
</template>
```

```
<script lang="ts">
```

```
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State, Getter, Action, Mutation, namespace } from 'vuex-class'
```

```
const {{ sentenceCase name }}State = namespace('{{ name }}', State)
const {{ sentenceCase name }}Mutation = namespace('{{ name }}', Mutation)
```

```
@Component({
```

```
  name: '{{sentenceCase name}}Page'
```

```
})
```

```
export default class {{ sentenceCase name }}Page extends Vue {
```



```
}  
</script>
```

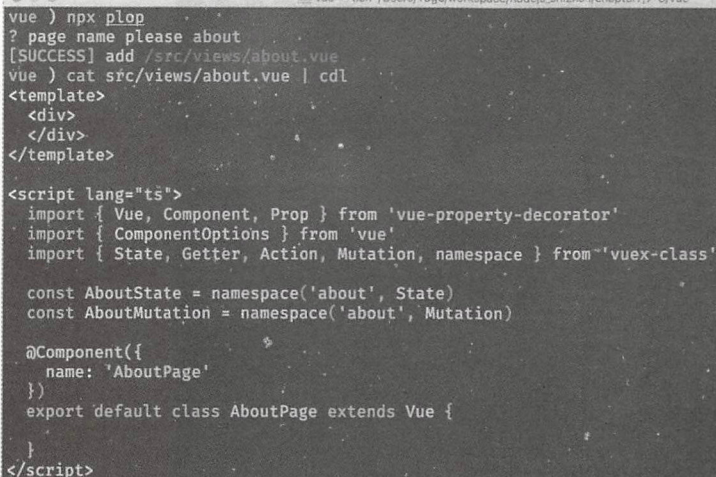
```
<style lang="sass" scoped>  
</style>
```

模板使用的是 handlebars 引擎, sentenceCase 是首字母大写的函数, plop 是内置的参数, 假如读者没有安装 vuex-class, 则先安装 vuex-class。

- 运行命令

```
npx plop
```

结果如图 5-11 所示:



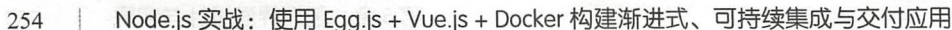
```
vue ) npx plop  
? page name please about  
[SUCCESS] add /src/views/about.vue  
vue ) cat src/views/about.vue | cdl  
<template>  
  <div>  
  </div>  
</template>  
  
<script lang="ts">  
  import { Vue, Component, Prop } from 'vue-property-decorator'  
  import { ComponentOptions } from 'vue'  
  import { State, Getter, Action, Mutation, namespace } from 'vuex-class'  
  
  const AboutState = namespace('about', State)  
  const AboutMutation = namespace('about', Mutation)  
  
  @Component({  
    name: 'AboutPage'  
  })  
  export default class AboutPage extends Vue {  
  
  }  
</script>
```

图 5-11

4. 异步加载

- 修改 router/index.ts

```
const Index = () =>  
  new Promise((resolve, reject) => {  
    require.ensure(['../views/index.vue'], require => {  
      resolve(require('../views/index.vue'))  
    })  
  })
```

```

  })

  export default function createRouter() {
    return new Router({
      mode: 'history',
      fallback: false,
      scrollBehavior: () => ({ y: 0, x: 0 }),
      routes: [{ path: '/', component: Index }]
    })
  }
}

```

这是目前笔者能找到的唯一的加载方式，不要认为 `../views/index.vue` 只是一个变量，就可以封装出一个方法。假如这样做，则服务端的 JSON 如下所示，`const I =` 就是封装出来的方法，在服务端会报错，`require.ensure` 没有这个方法，在 `Node.js` 里确实没有 `require.ensure`。

```

x.ts  TS app.ts  TS server.ts  vue-ssr-server-bundle.json  ...
Component.exports | views  第 3 个 (共 4 个)  ← → ≡ ×
"import Vue, { AsyncComponent } from 'vue'\nimport Router from
'vue-router'\nimport Hello from '../components/Hello.vue'\nimport
HelloDecorator from '../components/HelloDecorator.vue'\nimport {
resolve } from 'url'\nimport asyncComponent from
'../utils/import'\n\nVue.use(Router)\n\nconst Cool = () =>\n  new
Promise( (resolve, reject) => {\n    require.ensure(
['../components/Cool.tsx'], require => {\n      resolve(require
('../components/Cool.tsx'))\n    })\n  })\n  const I = (id = '') =>\n
new Promise( (resolve, reject) => {\n    require.ensure(id, require
=> {\n      resolve(require(id))\n    })\n  })\n  const Index = () => I
('../views/index.vue')\n\n// const Index = () =>\n//   new Promise(
(resolve, reject) => {\n//     require.ensure(['../views/index.vue'],
require => {\n//       resolve(require('../views/index.vue'))\n//
    })\n//   })\n\nexport default function createRouter() {\n  return
new Router({\n    mode: 'history',\n    fallback: false,\n    scrollBehavior: () => ({ y: 0, x: 0 }),\n    routes: [\n      {
path: '/', component: Index },\n      { path: '/cool', component:
Cool },\n      { path: '/hello/:id?', component: HelloDecorator }\n    ]
  })\n}"

```

也不要使用 `import`，因为 TypeScript 默认转义动态 `import`，而且在 TypeScript 中，`import` 在 ES6 模块模式下无法工作。哪怕修改成 `commonjs` 模块，经过转义后在 `client` 端就不会正常工作了。假如你阅读了源代码，会发现在 `utils` 中有一个 `import`，笔者希望通过 JS 绕过 TypeScript，但是 `import` 又会被 `webpack` 转换成 `webpackrequire`，所以服务端又找不到。

对于封装成方法无法工作的问题，笔者猜测可能使用了语法分析，对于动态计算的模块 ID 则无法分析出来。



```
Built at: 2018-4-2 15:31:58
Asset      Size  Chunks  Chunk Names
app.js     2.45 MiB    app [emitted] [big] app
0.cefbce3f5988103fe155.bundle.js 73.5 KiB    0 [emitted]
app.4bfd308d6ff13df749ae.hot-update.js 5.13 KiB    app [emitted]
4bfd308d6ff13df749ae.hot-update.json 45 bytes    [emitted]
vue-ssr-client-manifest.json 3.31 KiB    [emitted]
Entrypoint app [big] = app.js app.4bfd308d6ff13df749ae.hot-update.js
[./node_modules/es6-promise/dist/es6-promise.js] 28.6 KiB {app}
```

异步组件打包完成后，在终端里应该可以看到 0.xxxx.bundle.js 文件。

5. 新建 API 目录

按照下面的格式新建好目录：

```
src
├── api
│   └── v1
│       ├── client
│       │   └── index.ts
│       ├── server
│       │   ├── index.ts
│       │   └── IApi.ts
│       └── index.ts
```

目录里的文件现在暂时导出一些变量用于测试，IApi.ts 的用途是写 interface，因为我们通过路径重写，导致 API 的类型丢失了，所以只能再写一份，暂时没有什么更好的解决方法。

把 API 挂载到 Vue 实例上：

```
import api from './api/v1'

export default function createApp() {
  Vue.prototype.$api = api
}
```

在 entry 文件的 asyncData 里再添加 API 参数，记得导入 API。

- entry-client.ts

```
asyncDataHooks.map(hook => hook({ store, route: to, api }))
```

- entry-server.ts

```
if (asyncData) {
  return asyncData({
    store,
    route: router.currentRoute,
    api
  })
}
```



6. 开启插件

新建 `plugins.ts` 文件，专门用来引入插件。假如读者遇见了 `window undefined` 之类的错误，则可以使用 `jsdom` 模拟浏览器环境。

```
import Vue from 'vue'

import VueLazyload from 'vue-lazyload'
Vue.use(VueLazyload, {
  error: 'http://via.placeholder.com/450x190?text=404',
  loading: 'http://via.placeholder.com/450x190?text=loading&color=#368',
  attempt: 1,
  observerOptions: {
    rootMargin: '0px',
    threshold: 0.2
  }
})

import 'aos/dist/aos.css'

import aos from 'aos'

aos.init()
```

在 `app.ts` 里面导入：

```
if (isClient) {
  require('./plugins')
}
```

笔者宁愿 DOM 不匹配，也不想看见 `window undefined`，遇到问题时再调试。其实 SSR 的成本一点都不低，笔者宁愿写模板渲染也不想写 SSR，特别是 TypeScript SSR，但是有的时候没有选择。

7. 新建数据存储器

修改 `src/store/index.ts`。

添加模块：

```
import index from './modules/index'
const options: StoreOptions<State> = {
```

```
modules: {  
  index  
}
```

新建 `src/store/modules/index/index.ts`。

```
import { Module } from 'vuex'  
  
import { State as RootState } from '../..  
  
export class State {  
  images: any[] = []  
}  
  
const Index: Module<State, RootState> = {  
  namespaced: true,  
  state: new State(),  
  mutations: {  
    PUSH_IMAGES: (state, payload) => {  
      state.images = state.images.concat(payload)  
    }  
  }  
}  
  
export default Index
```

- 函数式编程的版本

抛开各种函数式规范，我们来实现一下简单的透镜。透镜是操作数组、对象的一种方式，就像 `getter` 和 `setter` 一样。通过 `len` 创建好透镜之后，可以通过 `set` 获取值和设置值，`over` 用于对值进行处理。

- 实现的透镜库

```
const len = (getter, setter) => ({  
  getter,  
  setter  
})  
  
const set = (len, val, states) => {
```



```
len.setter(val, states)
}
```

```
const get = (len, states) => {
  len.getter(states)
}
```

```
const over = (len, fn, states) => {
  const state = fn(len.getter(states))
  len.setter(state, states)
}
```

• 编写

```
const imagesLen = len(
  states => states.images,
  (payload, states) => {
    const newState = states.images.concat(payload)
    Vue.set(states, 'images', newState)
  }
)
```

```
const ImagesLen = R.lensProp('images')
over(imagesLen, R.concat(payload), state)
```

当然如果大家感兴趣，也可以看一下 `vuex-lens` 这个库。我们用这个库来改造一下。后端这个库可能会报错，因为这个库的格式不是一个标准的 Vue 插件格式，Vue 库导入的方式不对，作者用的是 `import`，所以大家可以把 `index.ts` 文件复制到 `shims` 目录下。

```
import { propLens, over } from '../../shims/vuex-lens'
const imagesLen = propLens('images')
```

```
over(imagesLen, R.concat(payload), state)
```

这个库多提供了一些帮助方法，比如 `propLens`，基于属性名创建透镜，并且内部也是基于 `Functor` 规范实现的，不依赖于 `Ramda`，下面阅读一下核心代码。

```
function lens<T>()
```

```

getter: (obj: any) => T,
setter: (v: T, obj: object) => any
) {
  return (toFunctorFn: (x: T) => IFunctor<T>): ((obj: object) => IFunctor<T>) => {
    return (obj: object): IFunctor<T> => {
      return toFunctorFn(getter(obj)).map((v: T) => {
        return setter(v, obj);
      });
    };
  };
}

```

第一眼看去，大多数人会非常头疼，我们把类型剔除。

```

function lens<T>(  
  getter,  
  setter  
) {  
  return toFunctorFn => {  
    return obj => {  
      return toFunctorFn(getter(obj)).map((v) => {  
        return setter(v, obj);  
      });  
    };  
  };  
}

```

这里的 `map` 跟数组的 `map` 没多大关系，这里的 `v` 其实就是 `getter(obj)` 的返回值，然后进行修改。而是否修改取决于你是否通过 `over` 调用透镜，因为 `over` 里传递了一个回调对值进行修改。

这里的 `map` 方法其实就是 `functor` 规范，它的接口如下。这个 `f` 其实就是对 `value` 值是否修改的一个回调。

```

interface IFunctor<T> {  
  value: T;  
  map: (f: (item: T) => any) => IFunctor<T>;  
}

```

创建 Functor 有两个方法，Const 会忽略处理函数，Identity 则不会。

```
function Const<T>(x: T): IFunctor<T> {
  return {
    value: x,
    map: (f: (item: T) => any) => Const<T>(x) // 忽略 f 函数
  };
}
```

```
function Identity<T>(x: T): IFunctor<T> {
  return {
    value: x,
    map: (f: (a: T) => any) => Identity<T>(f(x))
  };
}
```

最后我们看一下 set、get 和 over。

```
function set<T>(len: ILenses<T>, value: T, obj: any): any {
  const functor: IFunctor<T> = len((x: T) => Identity<T>(value))(obj);
  return obj;
}
```

首先向 len 中传递一个 $(x: T) \Rightarrow \text{Identity}(\text{value})$ 函数，它其实就是 toFunctorFn，因为这里是 Identity，所以才会执行 len 中 map 的 setter 方法。

```
function get<T>(len: ILenses<T>, obj: any): any {
  const functor: IFunctor<T> = len((x: T) => Const<T>(x))(obj);
  return functor.value;
}
```

get 的 toFunctorFn 是 Const，所以 map 里会忽略回调。

```
function over<T>(len: ILenses<T>, fn: (x: T) => T, obj: any): any {
  const functor: IFunctor<T> = len((x: T) => Identity<T>(fn(x)))(obj);
  return functor.value;
}
```

over 先调用当前的处理函数，再把值传递给后面的 setter。

8. 为什么需要数据存储器 store

首先我们理解一下 SSR, SSR 的意思就是服务端渲染, 为什么需要 SSR? 因为需要良好的 SEO。要 SEO, 我们直接用 HTML 不就好了吗? 但是普通的 HTML 渲染之后如果又需要保持高性能使用虚拟 DOM, 则需要单独再写一份前端代码来使用虚拟 DOM。也就是说, 后端一份初始的代码用于 SEO, 隐藏之前的初始模板, 然后在前端新建 Vue 实例挂载节点, 使用虚拟 DOM 更新, 这样也可以有良好的 SEO, 但是需要写两份代码。

其实 Vue 里就有 template 模板, 只不过 Vue 把它编译成 render 函数, 这个函数其实跟后端渲染引擎的 render 函数一样, 需要初始数据才能渲染出 HTML 模板, 所以我们需要解决初始数据的问题。所以我们才会在后端调用 asyncData 来获取数据。

当我们有初始数据的时候, 经过渲染会得到 HTML 模板并显示给前端, 但是这时又会出现一个问题, 已经有的数据, 如何在前端获得呢? 难道再获取一次? 于是就有了混合, 所以我们的数据才需要存储到全局状态里。在 SSR 环境中, Vuex 是用来做数据存储的。

9. 修改 index.vue

添加模板:

```
<div>
  <nav class="navbar is-black controll-header">
    <div class="container">
      <div class="navbar-menu">
        <div class="navbar-center">
          // ...
        </div>
        <div class="navbar-end is-hidden-touch">
          <a id="type-change-button" class="navbar-item" @click="changeType">
            <i class="fas fa-th-large" v-if="type==='large'"></i>
            <i class="fas fa-th" v-if="type==='normal'"></i>
          </a>
        </div>
      </div>
    </div>
  </nav>

  <div class="container" v-lazy-container="{ selector: 'img' }">
    <ul class="columns is-multiline img-box">
      <li v-for="(image,index) in images" v-bind:key="index" class="column">
```



```

data-aos="zoom-in" :class="['is-' + size, 'aos-init']">
  
</li>
</ul>
<div class="not-more" v-if="noMore">
  <article class="message">
    <div class="message-body">
      <h2 class="title is-4">莫得啦~</h2>
      <p>更多的画作等待您去创作。</p>
    </div>
  </article>
</div>
</div>
</div>

```

navbar-center 的部分省略了，可以在源码里找到，对于查询相关的功能笔者就不讲解了，先搭建骨架。

- 添加样式

特别要注意的是，.controll-header 的下边距为 1em，假如为 0 或者负值会造成 navbar-center 下拉部分异常，因为 z-index 的关系，被盖住了。

```

.controll-header
  margin: -1em -1em 1em -1em
.navbar-link
  color: #fff
.navbar-center
  margin-left: auto
  padding-top: 8px
  button.button
    background: #000
    color: #fff
    border-color: #000
.dropdown-content
  background: #000
  a.dropdown-item
    color: #fff

```

```

    &:hover
      background: #333
// 覆盖默认的风格，提高优先级
// .navbar.is-black .navbar-end .navbar-item
#type-change-button
  cursor: pointer
  &:hover
    background: #333

// 修正 nav 在 touch 终端隐藏的风格
+touch
  .navbar-menu
    display: flex
    background: #000
    padding: 0
    box-shadow: none
  .navbar-center
    margin-right: auto // 靠左，右边自动
    margin-left: .7em
  // .navbar-end
  //   margin-left: auto // 靠右，左边自动
  //   padding-right: .7em
  //   padding-top: 8px
  // .navbar-item
  //   color: #fff

.img-box
  padding: 12px 0
  img
    width: 100%
    object-fit: cover

```

• 修改逻辑

通过 `vuex-class`，我们可以映射全局状态中的数据和办法。

`type` 是用来控制显示的，比如一行显示 2 个，或者 4 个。

`activeIndex` 用来控制 `navbar-center` 里的 `hover`，笔者在写例子的时候，发现下拉栏的异常，

希望通过延迟添加 `is-active` 来解决问题，最后发现是 CSS 层级的问题，而这些方法就没有删除。

在 `asyncData` 里模拟了一些假的数据，下一节再添加真正的数据。

关于下拉栏获取数据的逻辑在 `mounted` 里，把一些代码包裹在不是 `$isServer` 的环境中，在服务端不会调用 `mounted`，不包裹其实也没什么问题。

最开始笔者通过监听滚动来实现下拉时间，但是在使用滚动时，为了实现更好的效果，可能要用函数去抖动拦截一下频繁触发的函数，而且还要兼容移动端，因为移动端的 `footer` 高度会变得更高，这样比较麻烦。可以使用 `IntersectionObserver` 来控制获取数据，逻辑的意思就是当 `footer` 进入可视区的时候加载数据。`intersectionRatio` 表示进入可视区的百分比。

对于 `changeType` 也有一个棘手的问题，AOS 会往区块上添加 CSS 样式类，当 `template` 更新的时候，会把这些 CSS 样式给冲掉，导致动画异常。

笔者的第一反应是把这些 AOS 的 CSS 添加到 `template` 里，但是如果这样，已经载入过的就会没有动画。笔者又去尝试修改 DOM 的 `class`，其实不建议修改 DOM 的 `class`，因为会像之前一样，一旦数据更新，会丢失之前的修改。因为只修改了 DOM，而没有修改状态，所以加载的新数据还是会渲染与之前一样的列数。

触发 AOS 初始化有两种方式，一种是触发滚动，另一种是调用它的 `init` 函数或者 `refresh` 函数。

`noMore` 是用来表示没有更多数据的提示。

```
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State, Getter, Action, Mutation, namespace } from 'vuex-class'
import { API } from '../api/v1/IApi'

const IndexState = namespace('index', State)
const IndexMutation = namespace('index', Mutation)

@Component({
  name: 'IndexPage'
})
export default class IndexPage extends Vue {
  @IndexState('images') images
  @IndexMutation('PUSH_IMAGES') Push
  public type = 'normal'
  public activeIndex = 0
```

```

public timer: any
public noMore = false

dropdownHover(index, event) {
  this.clearTimer()
  this.activeIndex = index
}

get size() {
  if (this.type == 'normal') {
    return 3
  }
  return 6
}

clearHover(event) {
  this.timer = setTimeout(() => {
    this.activeIndex = 0
  }, 300)
}

clearTimer() {
  if (this.timer) {
    clearInterval(this.timer)
  }
}

changeType() {
  // const els: HTMLElement[] = Array.from(
  //   document.querySelectorAll('.img-box:column')
  // )
  // if (els[0].classList.contains('is-3')) {
  //   els.forEach(el => {
  //     el.classList.remove('is-3')
  //     el.classList.add('is-6')
  //   })
  // } else {
  //   els.forEach(el => {

```



```

//     el.classList.remove('is-6')
//     el.classList.add('is-3')
//   })
// }

this.$nextTick(() => {
  const AOS = require('aos')
  setTimeout(function() {
    // AOS.refresh()
    const top =
      document.documentElement.scrollTop || document.body.scrollTop
    window.scrollTo(0, top + 1)
    window.scrollTo(0, top)
  }, 100)
})

if (this.type === 'normal') {
  this.type = 'large'
  return
}
this.type = 'normal'
}

destroy() {
  this.clearTimer()
}

async asyncData({ store }) {
  const images = Array(32).fill('http://via.placeholder.com/350x150')
  store.commit('index/PUSH_IMAGES', images)
}

loadMore() {
  if (this.noMore) {
    return
  }

  if (this.images.length > 120) {
    this.noMore = true
  }
}

```

```

    return
  }
  const images = Array(32).fill('http://via.placeholder.com/350x150')
  this.$store.commit('index/PUSH_IMAGES', images)
}
mounted() {
  // if (!this.$isServer) {
  //   const clientWidth = document.body.clientWidth
  //   window.onscroll = event => {
  //     const top =
  //       document.documentElement.scrollTop || document.body.scrollTop
  //     const allHeight = document.body.scrollHeight - window.innerHeight
  //     if (clientWidth < 600 && allHeight - top < 1000) {
  //       console.log('load')
  //       this.loadMore()
  //       return
  //     }
  //     if (clientWidth > 600 && allHeight - top < 400) {
  //       console.log('load')
  //       this.loadMore()
  //       return
  //     }
  //   }
  // }

  if (!this.$isServer) {
    var intersectionObserver = new IntersectionObserver(entries => {
      if (entries[0].intersectionRatio <= 0) return
      this.loadMore()
      console.log('Loaded new items')
    })

    intersectionObserver.observe(document.querySelector('footer.footer'))
  }
}

```

其实不少代码都有提升的空间，对于 `changeType` 简单的写法为：

```
changeType() {  
  this.type = this.type == 'normal' ? 'large' : 'normal'  
  this.AOSInit()  
}  
  
async AOSInit() {  
  await this.$nextTick()  
  const AOS = require('aos')  
  AOS.init()  
}
```

10. 效果

在移动端的效果如图 5-12 所示。“没有更多”的效果如图 5-13 所示。

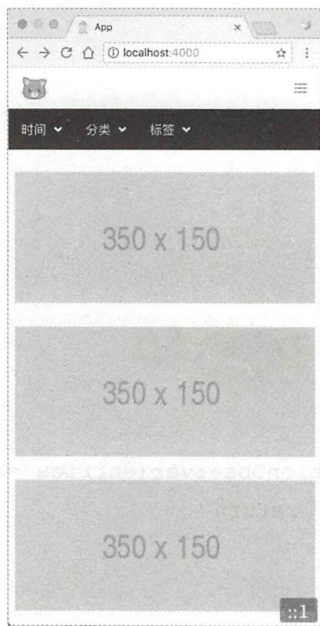


图 5-12



图 5-13

多列模式的效果如图 5-14 所示。

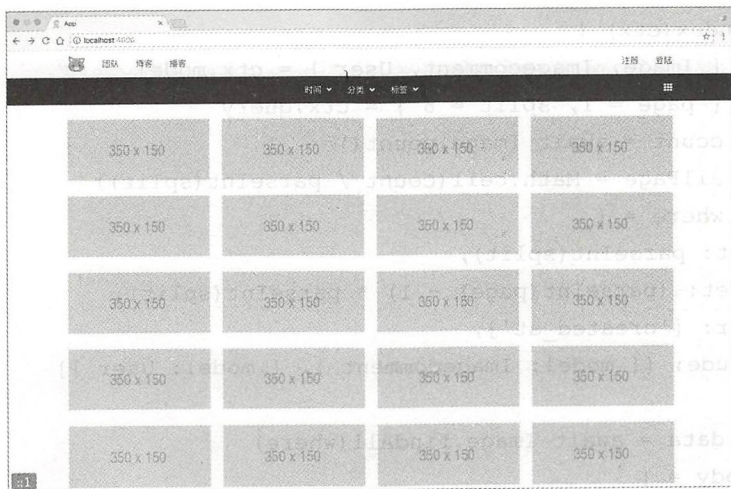


图 5-14

大图模式的效果如图 5-15 所示。

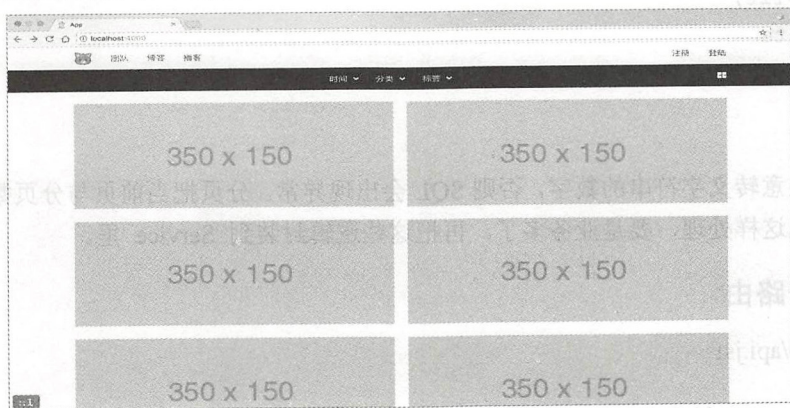


图 5-15

5.4 替换成为真实数据

5.4.1 完成后端 Image API

1. 添加控制器

修改 `app/controller/images.js`:


```
async index(ctx) {
  const { Image, Imagecomment, User } = ctx.model
  const { page = 1, split = 8 } = ctx.query
  const count = await Image.count()
  const allPage = Math.ceil(count / parseInt(split))
  const where = {
    limit: parseInt(split),
    offset: (parseInt(page) - 1) * parseInt(split),
    order: ['created_at'],
    include: [{ model: Imagecomment }, { model: User }]
  }
  const data = await Image.findAll(where)
  ctx.body = {
    allPage,
    currentPage: parseInt(page),
    split,
    data
  }
}
```

一定要注意转义字符串的数字，否则 SQL 会出现异常。分页把当前页与分页数相乘的积作为偏移量。先这样处理，要是业务多了，再把这些逻辑封装到 Service 里。

2. 注册路由

修改 app/api.js:

```
get: {
  '/image': ctl.image.index,
}
```

3. 用 PostMan 测试一下

当然读者要自己添加一些测试的数据，如图 5-16 所示。

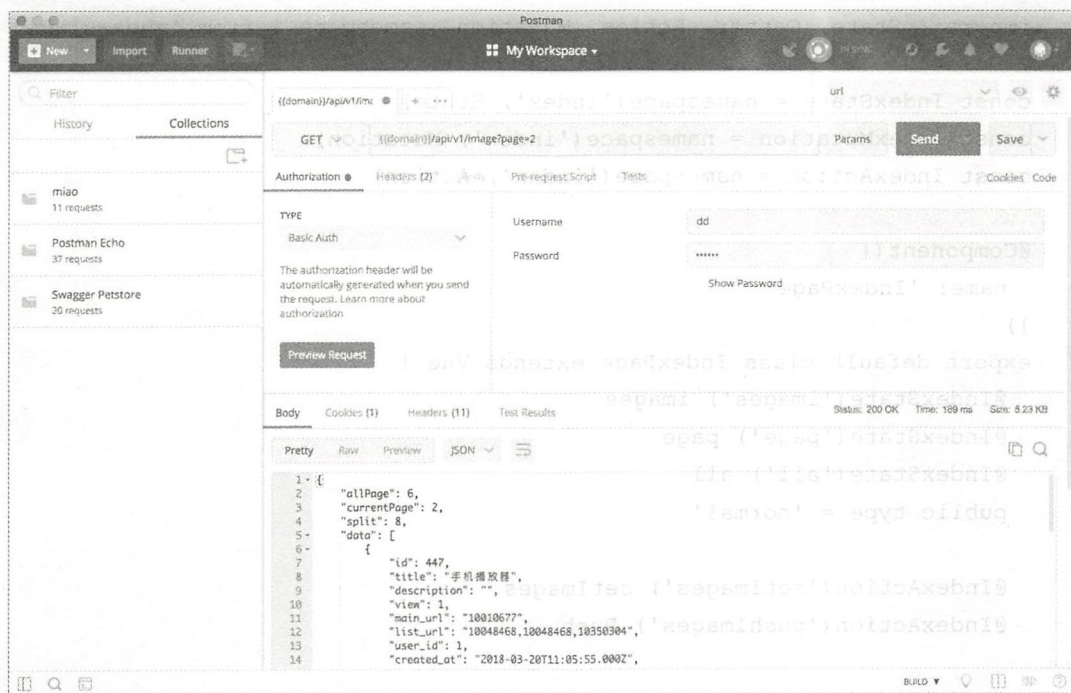


图 5-16

5.4.2 修改首页的代码

1. 页面逻辑

这一次把一些无用的代码和多余的解决方案给剔除了，这样大家见到的代码也会更清爽一些。

这里的一些定义文件，还是由于之前提到的问题，TypeScript 本身的语言服务跟 Vue 有一些冲突，也有可能是这两个 tsconfig 文件造成的，总之先使用 any 忽略掉，但是使用 any 的就没了类型，感觉使用 TypeScript 多此一举，笔者也很矛盾，只能期待它们越来越智能，直到没有 any 的那一天。

asyncData 尽量不要依赖组件中的方法，因为 asyncData 绑定的 this 有问题，utils 绑定的就是匹配出来的 Component，但这个 Component 是没有实例化的。所以解决问题的根源就是不依赖。

使用 vuex-class 可以让代码更简洁：

```

import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'

```

```
import { State, Getter, Action, Mutation, namespace } from 'vuex-class'

const IndexState = namespace('index', State)
const IndexMutation = namespace('index', Mutation)
const IndexAction = namespace('index', Action)

@Component({
  name: 'IndexPage'
})
export default class IndexPage extends Vue {
  @IndexState('images') images
  @IndexState('page') page
  @IndexState('all') all
  public type = 'normal'

  @IndexAction('setImages') setImages
  @IndexAction('pushImages') Push

  get size() {
    if (this.type === 'normal') {
      return 3
    }
    return 6
  }

  get noMore() {
    return this.all <= this.page
  }

  async changeType() {
    this.type = this.type === 'normal' ? 'large' : 'normal'
    this.AOSInit()
  }

  async AOSInit() {
    await this.$nextTick()
    const AOS = require('aos')
    AOS.init()
  }
}
```

```
}

async asyncData({ store, api }) {
  const { data: images } = await api.getImages()
  store.dispatch('index/setImages', images)
}

async loadMore(this: any) {
  if (this.noMore) {
    return
  }
  try {
    const { data: image } = await this.$api.getImages(this.page + 1)
    this.Push(image)
  } catch (e) {
    console.error(e)
  }
}

mounted(this: any) {
  const intersectionObserver = new IntersectionObserver(entries => {
    if (entries[0].intersectionRatio <= 0) return
    this.loadMore()
    console.log('Loaded new items')
  })
  intersectionObserver.observe(document.querySelector('footer.footer'))
}
```

2. 添加样式

我们还是向 Dribbble 的样式靠拢，对于用户头像的 hover，Dribbble 是用 JavaScript 写的，这里我们简单地用 CSS 代替，因为层级的关系，有可能出现问题。

笔者在调试的时候给 .column 添加了 z-index，不小心创建了一个层叠上下文，导致出现超出常理的显现，以及盖住了 hover 出来的层，如图 5-17 所示。

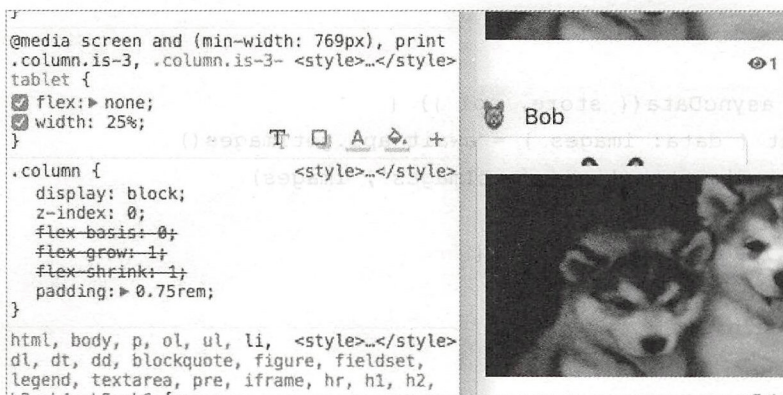


图 5-17

可以查看《CSS 世界》中的例子 (<http://demo.cssworld.cn/7/5-1.php>), 对于书写 CSS 力不从心的读者可以阅读 CSS 专家张鑫旭的《CSS 世界》。

```
.column.is-6 .main
```

```
height: auto !important
```

```
+touch
```

```
.img-box .main
```

```
height: auto !important
```

```
.img-box
```

```
padding: 12px 0
```

```
.main
```

```
width: 100%
```

```
background-color: #fff
```

```
background-size: cover
```

```
border-color: none
```

```
height: 150px
```

```
padding: 0.4em 0.4em 0
```

```
.info
```

```
background: #fff
```

```
text-align: right
```

```
padding: 0.2em 0
```

```
i::before
```

```
padding-right: .2em
```

```
.tag
```

```

    background:#fff
  .author
    margin-top: .6em
  .avatar
    display: flex
    font-size: 1em
    img
      width: 1.5em
      height: 1.5em
      margin-right: 10px
      border-radius: 50%

```

3. 添加模板

图片都是笔者自己准备的壁纸，读者可以去源码里寻找。懒加载的这个 Vue 框架其实还有不少 bug，这种使用方式是 bug 最少的一种。假如真正上线，则还需要考究。

```

<ul class="columns is-multiline img-box">
  <li v-for="(image,index) in images" :key="image.id" class="column" :class=
    "['is-' + size]">
    <img class="main is-block" v-lazy="'http://127.0.0.1:7001/public/
    download/'+image.main_url+'',1920,1080.jpg'"></img>
    <div class="info">
      <span class="tag"><i class="fas fa-eye">{{ image.view }}</i></span>
      <span class="tag"><i class="fas fa-comment">{{ image.ImageComments.
length }}</i></span>
      <span class="tag"><i class="fas fa-heart">22</i></span>
    </div>
    <div class="author">
      <div class="dropdown is-hoverable">
        <div class="dropdown-trigger">
          <a href="#" class="avatar">{{ image.User.username }}</a>
        </div>
        <div class="dropdown-menu">
          <div class="dropdown-content content has-text-centered">
            

```

```
<div class="buttons has-addons is-centered">
  <span class="button is-danger">关注</span>
  <span class="button">私信</span>
</div>
<p>我是{{ image.User.username }}, 欢迎关注我</p>
</div>
</div>
</div>
</li>
</ul>
```

5.4.3 添加 API 逻辑

1. 新建 api/v1/common.ts

公用的逻辑如下：

```
import axios from 'axios'
const http = axios.create({
  baseURL: 'http://localhost:7001/api/v1'
})
export function getImages(page = 1, limit = 8) {
  return http.get('/image', {
    params: {
      page,
      split: limit
    }
  })
}
```

2. 混合

api/v1/server/index.ts:

```
import * as common from '../common'
const info = {
```

```

    name: 'server',
    ...common
  }
export default info

```

api/v1/client/index.ts:

```

import * as common from '../common'

const info = {
  name: 'client',
  ...common
}
export default info

```

3. 修改 store/index

初始化的方法和追加的方法应该是不一样的，这时我们还可以记录一下页数和页码：

```

import { Module } from 'vuex'
import Vue from 'Vue'
import { State as RootState } from '../..'

import { propLens, over, set } from '../../../shims/vuex-lens'
const imagesLen = propLens('images')

export class State {
  images: any[] = []
  all = 1
  page = 1
}

const Index: Module<State, RootState> = {
  namespaced: true,
  state: new State(),
  mutations: {
    PUSH_IMAGES: (state, payload) => {
      over(imagesLen, (images: any) => R.concat(images, payload), state)
    },
  },
}

```



```
SET_IMAGES: (state, payload) => {
  set(imagesLen, payload, state)
},
ALL_PAGE: (state, payload) => {
  state.all = parseInt(payload)
},
PAGE: (state, payload) => {
  state.page = parseInt(payload)
}
},
actions: {
  setImages({ commit }, images) {
    commit('SET_IMAGES', images.data)
    commit('ALL_PAGE', images.allPage)
    commit('PAGE', images.currentPage)
  },
  pushImages({ commit }, images) {
    commit('PUSH_IMAGES', images.data)
    commit('ALL_PAGE', images.allPage)
    commit('PAGE', images.currentPage)
  }
}
}

export default Index
```

5.4.4 效果

这里，用户的个人简介字段没有添加，大家可以自行添加到模型里。头像的限制还是小了一点，因为笔者直接复制 Dribbble 用户的头像，URL 比较长，所以改成了 255 的长度（效果图略）。

5.5 图片详情页

本节我们将会用到一些第三方组件，会提供一些解决方案。

5.5.1 创建路由

修改 `src/router/index.ts` 里的代码：

```
const ImageDetail = () => {
  new Promise((resolve, reject) => {
    require.ensure(['../views/ImageDetail.vue'], require => {
      resolve(require('../views/ImageDetail.vue'))
    })
  })
}
```

在 `routes` 数组中添加路由：

```
{ path: '/image/:id', component: ImageDetail }
```

5.5.2 安装依赖

```
npm i vue-at vue-quill-editor photoswipe jsdom -S
```

`vue-at` 是用来@用户的组件，而 `vue-quill-editor` 则是提交评论的编辑器，`photoswipe` 是用来显示画廊的组件，`jsdom` 是在 Node 环境下模拟 DOM 的工具。

5.5.3 创建视图

可以用 `plop` 创建 `ImageDetail` 的视图文件，然后修改该文件。

1. 添加样式

```
<style lang="sass">
$pswp__assets-path : '../asserts/default-skin/'
@import '~photoswipe/src/css/main.scss'
@import '~photoswipe/src/css/default-skin/default-skin.scss'
</style>
```

在 SASS 里通常会有一些静态文件，默认路径是相对于该 SCSS 文件的路径，所以在项目里的文件通常会找不到，这里我们把 `default-skin` 复制到 `src/asserts` 目录下，然后指定 `$pswp__assets-path` 为相对于本文件的目录。一些第三方库文件引入顺序也非常重要，不要随意颠倒。

2. 添加逻辑

我们直接导入编译好的 photoswipe.js，而 ui-default 则是界面相关的 JS 文件。

galleryOptions 是初始化 photoswipe 的配置项，items 是显示在 photoswipe 的图片，showPhoto 则是显示图片的方法，根据官方给的例子，每次显示都需要创建一个新的实例。

this.\$ref 可以获取标记了相应元素的真实的 DOM 节点。

```
<script lang="ts">
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State, Getter, Action, Mutation, namespace } from 'vuex-class'
import AppComment from '@C/AppComment.vue'

import PhotoSwipe from 'photoswipe/dist/photoswipe.js'
import PhotoSwipeUI from 'photoswipe/dist/photoswipe-ui-default.js'

const ImageDetailState = namespace('ImageDetail', State)
const ImageDetailMutation = namespace('ImageDetail', Mutation)

@Component({
  name: 'ImageDetailPage',
  components: {
    AppComment
  }
})
export default class ImageDetailPage extends Vue {
  galleryOptions = {
    history: false,
    focus: false,
    shareEl: false,
    showAnimationDuration: 0,
    hideAnimationDuration: 0
  }

  items = [
    {
      src: 'https://farm2.staticflickr.com/1043/5186867718_06b2e9e551_b.jpg',
      w: 1024,
      h: 964
    }
  ]
}
```

```

    },
    {
      src: 'https://farm7.staticflickr.com/6175/6176698785_7dee72237e_b.jpg',
      w: 1024,
      h: 683
    }
  ]
  showPhoto() {
    let gallery = new PhotoSwipe(
      this.$refs.pswp,
      PhotoSwipeUI,
      this.items,
      this.galleryOptions
    )
    gallery.init()
  }
  mounted() {
    // build items array
  }
}
</script>

```

3. 添加模板

类名以 pswp 开头的都是 photoswipe 默认的初始模板，在 photoswipe 文档里可以找到，并且我们用 ref 做了标记，这样就可以从 this.\$refs 获取该节点真实的 DOM。main 区域则是显示的图片和简介，右边则是作者的信息，因为评论是一个公用组件，所以我们将其单独提取出来。

```

<div class="container">
  <div class="columns">
    <div class="column is-8">
      <div class="box">
        <div class="pswp" tabindex="-1" role="dialog" aria-hidden="true"
ref="pswp">
          <div class="pswp__bg"></div>
          <div class="pswp__scroll-wrap">

```



```

<div class="pswp__container">
  <div class="pswp__item"></div>
  <div class="pswp__item"></div>
  <div class="pswp__item"></div>
</div>

<div class="pswp__ui pswp__ui--hidden">

  <div class="pswp__top-bar">

    <div class="pswp__counter"></div>

    <button class="pswp__button pswp__button--close" title=
"Close (Esc)"></button>

    <button class="pswp__button pswp__button--share" title=
"Share"></button>

    <button class="pswp__button pswp__button--fs" title="Toggle
fullscreen"></button>

    <button class="pswp__button pswp__button--zoom" title="Zoom
in/out"></button>

    <div class="pswp__preloader">
      <div class="pswp__preloader__icn">
        <div class="pswp__preloader__cut">
          <div class="pswp__preloader__donut"></div>
        </div>
      </div>
    </div>

    <div class="pswp__share-modal pswp__share-modal--hidden
pswp__single-tap">
      <div class="pswp__share-tooltip"></div>
    </div>

    <button class="pswp__button pswp__button--arrow--left" title=

```

```

"Previous (arrow left)">
    </button>

    <button class="pswp__button pswp__button--arrow--right" title=
"Next (arrow right)">
    </button>

    <div class="pswp__caption">
        <div class="pswp__caption_center"></div>
    </div>

</div>

</div>
</div>

<div class="main">
    
</div>
<div class="list-img">
    
    
    
</div>

<div class="content">
    这是我家的狗狗，可爱不？
</div>
</div>
</div>
<div class="column is-4">
    <div class="box has-text-centered">
        

```

```

    <p class="username">username</p>
    <p class="desc small">desc</p>
    <p class="remote">
      <i class="fas fa-laptop"></i> 支持远程</p>
  </div>
</div>
</div>

<app-comment />
</div>

```

4. 组件样式

style 标签并非只能有一个，现在我们来添加组件样式。

```
<style lang="sass" scoped>
```

```
.box
```

```
  box-shadow: none
```

```
.avatar
```

```
  width: 68px
```

```
  height: 68px
```

```
  border-radius: 50%
```

```
  object-fit: cover
```

```
.username
```

```
  font-size: 1.5em
```

```
  margin: 10px 0 5px
```

```
.small
```

```
  font-size: .9em
```

```
.desc,
```

```
.remote
```

```
  margin-bottom: 10px
```

```
.list-img
```

```
  display: flex
```

```
  margin: 10px 0 20px
```

```
  img
```

```
    max-width: 24%
```

```
    margin-right: .5%
```

```

    object-fit: cover
    height: 100px
  </style>

```

5.5.4 添加插件

1. 修改 src/app.sass

现在添加评论编辑器的样式，因为我们没有在 `ts` 里添加处理 CSS 的配置，所以在 SASS 里配置，不过大家也可以自己进行配置。

```

@import '~quill/dist/quill.core.css'
@import '~quill/dist/quill.snow.css'
@import '~quill/dist/quill.bubble.css'

```

2. 修改 src/plugin.ts

我们使用编辑器 SSR 指令的版本，不过这里也可以直接引入组件，因为 `plugin.ts` 是 `isClient` 时才会载入。假如第二行在 `app.ts` 里使用时就会遇到问题。

```

import VueQuillEditor from 'vue-quill-editor/src/ssr'
// import VueQuillEditor from 'vue-quill-editor'
Vue.use(VueQuillEditor)

```

3. Hack DOM

假如因为某些原因，一定要在 `app.ts` 里引入呢？我们使用 JSDOM 模拟了一些浏览器的全局变量，让它工作。有时我们在组件内部引入的第三方库也会报出找不到 `window` 等问题，解决方法有两种：一种是 Hack DOM，另一种就是在全局中判断 `isClient`，在客户端环境全局注册组件。在源码中都有相应的注释，读者可以解开自行测试。

```

if (isServer) {
  const jsdom = require('jsdom')
  const dom = new jsdom.JSDOM('')
  ;(global as any).window = dom.window
  ;(global as any).document = dom.window.document
  ;(global as any).Node = dom.window.Node
  ;(global as any).navigator = dom.window.navigator
}

```



```
import VueQuillEditor from 'vue-quill-editor'
Vue.use(VueQuillEditor)
```

5.5.5 创建评论组件

1. 新建模板

修改 plopfile.js, 按照 page 的样子修改出一份 component 的样式。

```
plop.setGenerator('component', {
  description: 'new vue component',
  prompts: [
    {
      type: 'input',
      name: 'name',
      message: 'component name please'
    }
  ],
  actions: [
    {
      type: 'add',
      path: 'src/components/{{name}}.vue',
      templateFile: 'templates/page.hbs'
    }
  ]
})
```

2. 新建 AppComment.vue

添加模板

因为导入的是 SSR 版本, 我们使用 v-quill 指令创建评论输入框, 假如你是使用组件的形式, 那么先导入组件, 然后解开下面的注释即可, 笔者进行了测试, 都是可以成功的。使用 v-model 绑定数据的时候, VSCode 可能在 ESLint 中会报错, 官方的帮助文档给出了使用注释, 笔者已经加在代码中, 但是似乎并没有正常工作。所以就把验证禁用了, 并添加了 template 格式化的插件。

修改 VSCode 配置:

```
"vetur.format.defaultFormatter.html": "js-beautify-html",
"vetur.validation.template": false,
```

在回复按钮上面的子评论数里，绑定了单击事件，只有单击的时候才会显示子评论，同样单击“我要评论”按钮才显示评论框。目前来说输入的都是些假的数据，因为对接后台还需要一些依赖项，比如用户登录才能评论，所以我们先把“骨架”搭起来。

```
<div class="comment box is-clearfix">
  <h2>评论</h2>
  <ul>
    <li>
      <div class="comment-header">
        <span class="username">username</span>
        <span class="publish-date">2017-01-22</span>
      </div>
      <div class="content comment-content">
        <p>我是评论</p>
      </div>
      <div class="comment-footer has-text-right">
        <span @click="showComment(1)">
          <i class="fas fa-comments small"></i> 3</span>
        <span>
          <button>回复</button>
        </span>
      </div>
      <transition name="fade">
        <div class="child-comments" v-show="showIndex == 1">
          <div class="comment">
            <div class="comment-header">
              <span class="username">username</span>
              <span class="publish-date">2017-01-22</span>
            </div>
            <div class="content comment-content">
              <p>我是评论</p>
            </div>
          </div>
        </div>
      </div>
    </li>
  </ul>
</div>
```

```

        <div class="comment-header">
          <span class="username">username</span>
          <span class="publish-date">2017-01-22</span>
        </div>
        <div class="content comment-content">
          <p>我是评论</p>
        </div>
      </div>
      <div class="comment">
        <div class="comment-header">
          <span class="username">username</span>
          <span class="publish-date">2017-01-22</span>
        </div>
        <div class="content comment-content">
          <p>我是评论</p>
        </div>
      </div>
    </div>
  </transition>
</li>
</ul>
<!-- eslint-disable-next-line vue/valid-v-model -->
<!-- eslint-disable -->
<div class="comment-input has-text-centered">
  <button class="button is-small" v-if="!showInput" @click="showInput
= !showInput">我要评论</button>
  <transition name="fade">
    <div v-if="showInput">
      <at v-model="content" :members="members">
        <div class="quill-editor" v-model:content="content" v-quill:
myQuillEditor="editorOption">
          </div>
        </at>
        <button class="button is-small is-pulled-right" @click=
"pushComment">提交</button>
      </div>
    </transition>
  </div>

```

```
<!-- <quill-editor></quill-editor> -->
</div>
```

添加逻辑

导入 vue-at，这个组件暂时先定义 comments、author、type 属性。members 是用户可以@的字符串，而给用户添加消息，则需要我们在后端通过正则的方式提取。

默认子评论是隐藏的，只有当 showIndex 等于当前 v-for 列表的 index 时才会显示出来。同样输入表单也是隐藏的，只有当 showInput 为 true 时才会显示出来。

```
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import At from 'vue-at'

@Component({
  name: 'AppComment',
  components: {
    At
  },
  props: ['comments', 'author', 'type']
})
export default class CommentPage extends Vue {
  showIndex = 0
  content = ''
  editorOption = {
    theme: 'bubble',
    placeholder: '请输入评论'
  }
  members = ['223']
  showInput = false
  showComment(index) {
    if (this.showIndex == index) {
      this.showIndex = 0
      return
    }
    this.showIndex = index
  }
  mounted() {
```



```
// TODO: 将 Comments 和 Author 里面的用户提取出来赋值给 members
// TODO: 完善 Push 操作
}
pushComment() {
  // this.$api.pushComment('type', this.content)
}
```

5.5.6 测试

相比较设计的时候而言，我们添加了更多的功能。

- 默认情况
- 单击评论数
- 显示评论框

单击“我要评论”按钮，然后输入@，看是否有提示，确认 members 数组中有测试的数据。

5.5.7 关于服务端访问 DOM

在服务端始终没有浏览器的相关对象，访问时就会报错，虽然我们可以 Hack 解决问题，但是怎样才是正确的方式呢？

1. 把访问 DOM 的逻辑放在 mounted 或 beforeMount 中

在服务端调用渲染函数的目的其实就是想获得一些字符串而已，然后响应给前端。所以，在服务端渲染出来的组件全都是字符串，它并不需要挂载到任何地方，甚至不会销毁这些字符串，所以它只会调用 beforeCreate 和 created 函数。

2. 尽量尝试自己开发

将之前的 vue-at 组件导入为 src 里的源码，记得开启对 node_modules 里的 Vue 的编译，即去掉 exclude。

```
import At from 'vue-at/src/At.vue'
```

这时注释模拟的 DOM 代码，会发现它依然可以工作。因为打包出来的代码需要运行才能获得内容。

3. 使用指令

在服务端也可以提供指令的服务端版本，在 `createRenderer` 的时候当作配置项传递进去，比如 `v-show`，`VNodeWithData` 是 `flow` 类型，稍后会讲解。服务端的指令只许访问虚拟节点 `node`，所以服务端实现只是把样式对象推送到了 `data` 的 `style` 中。

更多的类型定义可以在以下地址中找到：

- <https://github.com/vuejs/vue/blob/62265035c0c400ad6ec213541dd7cca58dd71f6e/types/vnode.d.ts>;
- <https://github.com/vuejs/vue/blob/62265035c0c400ad6ec213541dd7cca58dd71f6e/flow/vnode.js>.

一种是 `flow` 的类型，另一种是 `TypeScript` 的类型，都差不多。

```
export default function show (node: VNodeWithData, dir: VNodeDirective) {
  if (!dir.value) {
    const style: any = node.data.style || (node.data.style = {})
    if (Array.isArray(style)) {
      style.push({ display: 'none' })
    } else {
      style.display = 'none'
    }
  }
}
```

其中 `node` 的定义为以下内容，`style` 是记录虚拟 DOM 本身样式的对象。

```
declare interface VNodeData {
  key?: string | number;
  slot?: string;
  ref?: string;
  is?: string;
  pre?: boolean;
  tag?: string;
  staticClass?: string;
  class?: any;
  staticStyle?: { [key: string]: any };
```

```
style?: Array<Object> | Object;
normalizedStyle?: Object;
props?: { [key: string]: any };
attrs?: { [key: string]: string };
domProps?: { [key: string]: any };
hook?: { [key: string]: Function };
on?: ?{ [key: string]: Function | Array<Function> };
nativeOn?: { [key: string]: Function | Array<Function> };
transition?: Object;
show?: boolean; // marker for v-show
inlineTemplate?: {
  render: Function;
  staticRenderFns: Array<Function>;
};
directives?: Array<VNodeDirective>;
keepAlive?: boolean;
scopedSlots?: { [key: string]: Function };
model?: {
  value: any;
  callback: Function;
};
};
```

而对于客户端的 `show` 指令，可以在 <https://github.com/vuejs/vue/blob/62265035c0c400ad6ec213541dd7cca58dd71f6e/src/platforms/web/runtime/directives/show.js> 上找到。

4. vue-quill-editor/src/ssr.js 是否真正运行在服务端

大家可能注意到了导入它的逻辑是在 `plugins.ts` 里的，只会在客户端运行。因为 `ssr.js` 导入了 `quill`，`quill` 访问了 `window`，就会失败，所以还是不能运行在服务端。这个 `ssr.js` 注册了一个客户端的指令，其实通过指令生成的 `Vue` 开发工具是无法识别的，因为它不是组件，如图 5-18 所示，只有 `At`。作者创建 `ssr.js` 的意图可能是为了兼容 `Nuxt.js`。



图 5-18

既然 `ssr.js` 里访问了 `quill`, 是不是可以忽略呢? 可以, 但是需要对 JS 开启 `node_modules` 编译。那么就相当于开发环境了, 就会缺少比较多的依赖, 笔者安装了比较多的依赖。由于 `vue-lazyload` 使用了 `babel-helpers`, 它又使用的是 `rollup` 编译, 开启了内置的 `helpers` 支持, 所以才会有一个 `external-helpers` 插件。最终笔者只好修改其中的 `babelrc`, 关闭 `external-helpers` 插件。当然结果是成功了, 但这太麻烦了, 对 `Node` 不熟悉的读者可能根本无法驾驭, 在实际开发中没必要这样。

- `webpack/server.ts` 配置

```
externals: [
  nodeExternals({
    whitelist:[ /\.css$/, /\.styl$/, /\.vue$/, /vue-quill-editor/]
  }),
  'quill'
],
```

- `rollup` 开启了 `helpers` 支持

```
const bundle = await rollup.rollup([
  input:path.resolve(__dirname, 'src/index.js'),
  plugins: [
    resolve(),
    commonjs(),
    babel({runtimeHelpers: true}),
    uglify()
  ]
])
```

- 因为我们使用自己的 `webpack` 编译, 所以需要关闭 `external-helpers` 插件。笔者尝试了手动导入 `babel-helpers`, 没有成功, 所以只好将 `.babelrc` 里的 `external-helpers` 删除,

否则会找不到 babelHelpers。

```
{
  "presets": [
    [
      "env",
      {
        "modules": false
      }
    ]
  ],
  "plugins": [
    ["external-helpers"]
  ]
}
```

- 安装所缺少的组件

```
> npm i babel-plugin-transform-object-assign babel-plugin-external-helpers -D
+ babel-plugin-transform-object-assign@6.22.0
+ babel-plugin-external-helpers@6.22.0
added 2 packages from 1 contributor in 31.438s
> npm i babel-helpers -D
+ babel-helpers@6.24.1
updated 1 package in 34.937s
> npm i assign-deep -D
+ assign-deep@0.4.7
added 3 packages from 9 contributors in 37.563s
```

5.6 注册页面

5.6.1 注册路由

修改 src/router/index.ts:

```
const signup = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/signup.vue'], require => {
      resolve(require('../views/signup.vue'))
    })
  })
```

把路由放到 routes 数组中:

```
{ path: '/signup', component: signup }
```

5.6.2 新建 signup.vue 页面

1. 添加模板

success 是存放成功消息的变量, error 则是存放失败消息的变量, 通过 class 绑定的对象语法来绑定特定的样式。

然后新建一些表单, 绑定相应的变量, 当单击“提交”按钮的时候就将数据提交到后端, 这里笔者没有做一些前端验证, 假如大家希望给用户更好的提示消息, 可以添加一些前端校验的逻辑。

```
<div class="has-text-centered">
  <div class="notification" :class="{ 'is-danger': error, 'is-success':
success}" v-if="error || success">
    <button class="delete" @click="error = '', success=''"></button>
    {{ error }} {{ success }}
  </div>
  <h2 class="title is-2">注册</h2>
  <p class="subtitle title">好不容易遇见你</p>
  <div class="box">
    <div class="field">
      <div class="control">
        <input class="input" type="text" v-model="email" placeholder="用
户邮箱">
      </div>
    </div>
    <div class="field">
      <div class="control">
        <input class="input" type="text" v-model="username" placeholder="
用户名">
      </div>
    </div>
    <div class="field">
      <div class="control">
```

```

        <input class="input" type="password" v-model="password"
placeholder="密码">
      </div>
    </div>
    <div class="field">
      <div class="control">
        <input class="input" type="password" v-model="confirm_password"
placeholder="确认密码">
      </div>
    </div>
    <div class="field">
      <div class="control">
        <input class="input" type="text" v-model="code" placeholder="邀请码">
      </div>
    </div>
    <a class="button is-black" @click="submit">
      注册
    </a>
  </div>
  <div class="box has-text-centered">
    <p>忘记密码了？ 那就
      <a href="#">找回密码</a> 吧！ </p>
    <p>已有账户？ 那就赶紧
      <a href="#">登录</a> 吧！ </p>
  </div>
</div>

```

2. 添加样式

对默认的样式做一些覆盖，调整边距：

```

.title
  margin-top: 2rem
  margin-bottom: 2rem

.box:first-of-type
  padding: 2rem

.field
  margin-bottom: 1rem

```



```
&:last-of-type
  margin-bottom: 1.5rem
.box
  width: 30%
+touch
  width: 50%
+mobile
  width: 70%
margin: 2rem auto
box-shadow: none
&:last-child p
  font-size: .9rem
a
  color: #000
  border-bottom: 1px solid
.button
  width: 40%
```

3. 添加 API

- 修改 `api/v1/client/index.ts`

下面的注释给出了几个函数式编程的例子，因为 `user` 是原样传递给 `http.post` 的，所以我们可以使用 `curry` 或者 `partial` 固定一个参数的位置。这在函数式里叫 Pointfree Style 风格函数，即无参数函数。

大家可以对比一下 `signUp` 的几种实现方式。

```
signUp : (user) => R.curry(http.post)('/signup')(user)
signUp : R.curry(http.post)('/signup') // 没有 user 参数
import * as common from '../common'
import axios from 'axios'
const http = axios.create({
  baseURL: 'http://localhost:7001/api/v1'
})

const info = {
  name: 'client',
  ...common,
  signUp(user) {
```




```

    return http.post('/signup', user)
  }
}
export default info

```

4. 添加逻辑

添加了一个空 `asyncData` 方法是为了触发加载条。还是由于编辑器的原因，`this` 上找不到 `$api`，在参数里指定 `this: any` 跳过该错误。

```

import { Vue, Component, Prop } from 'vue-property-decorator'

@Component({
  name: 'SignupPage'
})
export default class SignupPage extends Vue {
  username = ''
  email = ''
  password = ''
  confirm_password = ''
  code = ''
  error = ''
  success = ''
  asyncData() {}
  async submit(this: any) {
    try {
      this.error = ''
      const ret = await this.$api.signup(this.$data)
      if (ret) {
        this.success = '注册成功，请到邮箱激活你的账户后登录.'
      }
    } catch (e) {
      this.error = e.response.data
    }
  }
}

```



5.6.3 增强错误提示

修改后端 validator 的错误定制。当发生 SQL 错误的时候, 错误会在 errors 属性上。这样当 email 重复的时候就不会只出现一个 Validator Error 了。不过提示还是英文的, 可以自行在前端进行判断。

```
config.validator = {
  async formate(ctx, error) {
    info(['egg-y-validator'] -> %s', JSON.stringify(error, ' '))
    if (Array.isArray(error.errors)) throw new Error(error.errors[0].message)
    if (Array.isArray(error)) throw new Error(error[0].message)
    throw error
  }
}
```

5.7 登录页面

如果对 SPA 如何社会化登录比较困惑, 这一节我们就来解决这个问题。原理其实就是通过 VueRouter 拦截 Token, 将其存储到浏览器, 由于是通过 GET 请求的 query 方式传递 Token 的, 容易被历史记录, 所以可能不太安全, 后面会给出对于升级策略的一些思路。

如果 VSCode 报错, 忽略即可, 只要能正常编译, 就不用管它。

1. 安装依赖

```
npm i localfrage
```

localfrage 是浏览器存储的一个封装库, 提供了 Promise 风格的接口, 支持 WebSQL、IndexedDB、LocalStorage 等。我们使用 LocalStorage 持久地存储用户的 Token。

2. 添加路由

修改 src/router/index.ts:

```
const signin = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/signin.vue'], require => {
      resolve(require('../views/signin.vue'))
    })
  })
```





将路由添加到数组中：

```
{ path: '/signin', component: signin }
```

3. 页面

通过 plop 新建 signin 页面。

模板

与之前的注册页面类似，上面是通知的部分，下面则是左右分割的两栏，左边是 logo，右边是输入表单，上边还有 GitHub 登录。

```
<div class="container">
  <div class="notification has-text-centered" :class="{ 'is-danger': error,
'is-success': success}" v-if="error || success">
    <button class="delete" @click="error = '', success=''"></button>
    {{ error }} {{ success }}
  </div>
  <div class="columns">
    <div class="is-6 column has-text-centered">
      
      <h2 class="title is-2">欢迎回来...</h2>
    </div>
    <div class="column is-4 is-offset-2">
      <div class="box is-clearfix">
        <div class="top">
          <a class="button is-block is-black" href="http://024c6329.
ngrok.io/passport/github">
            <i class="fab fa-github"></i> Github 登录</a>
        </div>
        <div class="form">
          <div class="field">
            <div class="control">
              <input class="input" type="text" v-model="email" placeholder=
"用户邮箱">
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
```





```

<div class="field">
  <div class="control">
    <input class="input" type="password" v-model="password"
placeholder="密码">
  </div>
</div>
<button class="button is-pulled-right" @click="submit">登录</button>
</div>
<div class="box">
  <p class="has-text-centered">
    忘记密码了? 那
    <a href="#">找回密码</a>吧。<br/> 没有账户? 赶紧
    <a href="#">注册</a>吧! </p>
  </div>
</div>
</div>
</div>

```

样式

```

.top
  padding: .5rem 0 1rem
.box
  box-shadow: none
.box > button
  margin-top: 1rem
.box > p.has-text-centered
  font-size: .8rem
.logo
  width: 190px
  margin: 0 auto 2rem
.is-6.has-text-centered
  display: flex
  align-content: center
  flex-direction: column
  justify-content: center

```





页面逻辑

我们应该将 `beforeRouteEnter` 放到配置项里，否则可能找不到这个路由钩子，为了避免在服务端运行（服务端执行 `localfrage` 会报错），我们用 `isClient` 包裹一下，通过当前路由对象的 `to` 判断 `query` 上是否有 `Token`，有 `Token` 则通过 `localfrage` 存储起来，然后通过 `Token` 请求用户的数据。

假如 `query` 不存在，则判断浏览器存储中是否还有，如果有说明已经登录，直接跳转到首页即可。

在 `beforeRouteEnter` 里是访问不到 `this` 的，但是可以通过给 `next` 传递一个回调来获取 `vm`，这个 `vm` 就是 `this`。

一定要调用 `next`，否则后端和前端的路由就一直进不去，会一直处于“载入中”的状态。

```
import { Vue, Component, Prop } from 'vue-property-decorator'
import { State } from 'vuex-class'
import * as local from 'localforage'

@Component<Vue>({
  name: 'SigninPage',
  beforeRouteEnter(to, from, next) {
    if (isClient) {
      if (to.query.token) {
        local
          .setItem('user_token', to.query.token)
          .then(token => {
            return next((vm: SigninPage | any) => {
              vm.$api
                .getUser(token)
                .then(user_info =>
                  vm.$store.commit('USER_INFO', user_info.data.user)
                )
                .catch(console.error)
            })
          })
          .then(() => next('/'))
          .catch(next)
      } else {
        local
```





```

      .getItem('user_token')
      .then(token => {
        if (token) return next('/')
        next()
      })
      .catch(next)
    }
  }
})

export default class SigninPage extends Vue {
  password = ''
  email = ''
  error = ''
  success = ''
  @State('user_token') user_token
  @Mutation('USER_INFO') USER_INFO: any
  @Mutation('USER_TOKEN') USER_TOKEN: any

  async submit(this: any) {
    this.error = ''
    this.success = ''
    try {
      const { data } = await this.$api.signIn({
        email: this.email,
        password: this.password
      })
      await local.setItem('user_token', data)
      this.USER_TOKEN(data)
      this.success = '登录成功'
      const { data: { user } } = await this.$api.getUser(data)
      this.USER_INFO(user)
      setTimeout(() => {
        this.$router.push('/')
      }, 500)
    } catch (e) {
      this.error = e.response.data
    }
  }
}

```





4. 载入用户 Token

为了让用户每次刷新都能载入 Token，这里执行判断，假如全局状态上不存在数据，则从 localfrage 中取。有错误说明失效了，删除已经存在的数据。

修改 entry-client.ts:

```
localStorage.getItem('user_token').then(token => {
  if (token && !store.state.user_token) {
    store.commit('USER_TOKEN', token)
    api
      .getUser(token)
      .then(user_info => store.commit('USER_INFO', user_info.data.user))
      .catch(console.error)
  }
})
```

5. 添加发送请求 API

在 api/v1/client/index.ts 里添加方法。

当响应不是 200 时说明 Token 失效，可以删除它。

```
signIn(user) {
  return http.post('/signin', user)
},
getUser(token) {
  return http.get('/', {
    headers: {
      Authorization: 'Bearer ' + token
    }
  }).catch(error => {
    localStorage.removeItem('user_token')
  })
}
```

6. 添加全局状态

- 添加初始状态

token 用来存储令牌，info 用来存储用户信息。



```
export class State {
  user_token = null
  user_info = null
}
```

- 添加 mutations

```
mutations: {
  USER_TOKEN(state, payload) {
    state.user_token = payload
  },
  USER_INFO(state, payload) {
    state.user_info = payload
  }
},
```

7. 修改菜单栏

模板

```
.navbar-end
template(v-if="user_info")
  router-link.navbar-item(to="/me", v-text="user_info.username")
  a.navbar-item(v-on:click="logout") 注销
template(v-else)
  router-link.navbar-item(to="/signup") 注册
  router-link.navbar-item(to="/signin") 登录
```

退出逻辑

```
logout(this: any) {
  this.$store.commit('USER_TOKEN', '')
  this.$store.commit('USER_INFO', '')
  local.removeItem('user_token')
}
```

映射全局状态

在组件内部映射 user_info 的全局状态。

```
@State('user_info') user_info: object
```




8. 社会化登录

启动内网穿透

当然前提是启动了后端。

```
ngrok http 7001
```

效果如图 5-19 所示。

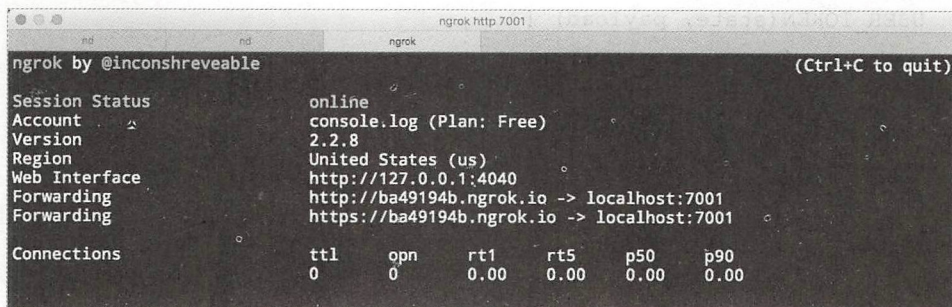


图 5-19

更新网址

进入你的 GitHub 的 Developer settings，如图 5-20 所示：

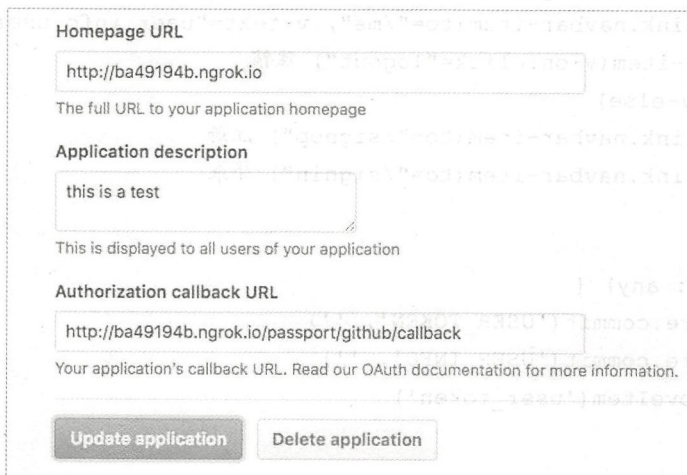


图 5-20

修改视图的链接

确保是从公网的域名访问的，使用 localhost 访问是会被拒绝的。



```
<div class="top">
  <a class="button is-block is-black" href="http://ba49194b.ngrok.io/passport/github">
    <i class="fab fa-github"></i> Github 登录</a>
</div>
```

修改后端 JWT

忽略一些不需要拦截的 API:

```
ignore: [
  /\passport/i,
  /\sign/,
  /\image/,
  /\email/,
  /\alipay/,
  /\admin/,
  /\.*\.(js|css|map|jpg|png|ico)/
]
```

修改重定向地址

配置 frontURL:

config.frontURL 为前端 SPA 的地址, 在 config.default.js 中进行配置。

```
config.frontURL = 'http://localhost:4000'
```

修改 app/passport/github.js:

我们需要通过 ctx.unsafeRedirect 进行跳转, 笔者尝试过设置安全的白名单, 但是似乎对于 localhost 的地址并没有生效, 我们直接用 ctx.unsafeRedirect 即可。

```
'use strict'
```

```
const R = require('ramda')
module.exports = async (ctx, user) => {
  const data = { uid: user.id, provider: user.provider }
  const auth = (await ctx.model.Auth.findOrCreate({
    where: data,
    default: data
  }))[0]
```



```
if (auth.user_id) {
  const existsUser = await ctx.model.User.findOne({
    where: { id: auth.user_id }
  })
  const raw_user = R.omit(['password'], existsUser.toJSON())
  const token = await ctx.sign_token(raw_user)
  ctx.body = token
  ctx.unsafeRedirect(ctx.app.config.frontURL + '/signin?token=' + token)
  return raw_user
}
// 调用 service 注册新用户
const newUser = await ctx.model.User.create({
  username: user.displayName,
  avatar: user.photo,
  email: user.profile.emails[0].value
})
auth.user_id = newUser.id
await auth.save()
const raw_user = R.omit(['password'], newUser.toJSON())
const token = await ctx.sign_token(raw_user)
ctx.body = token
ctx.unsafeRedirect(ctx.app.config.frontURL + '/signin?token=' + token)
return raw_user
}
```

9. 测试

GitHub 登录

直接单击 GitHub 登录。

当登录成功后，会自动跳到首页。

普通登录

输入错误的情况：

其实对于不存在的用户邮箱，会报出获取不到 password 的错误，原因是没有做空值判断，大家可以自行完善一下模型的 Auth 方法。



输入正确会跳转到首页。

10. 升级方案

无论是何种功能，一定要力求多种解决方案。

方案一

将请求 GitHub 的登录服务放在 SPA 的后端，通过 HTTP 请求让 API 的服务发放 `jwt_token`，然后存到 Cookie 中，这样就不会有任何数据出现在用户的浏览器上，如图 5-21 所示。

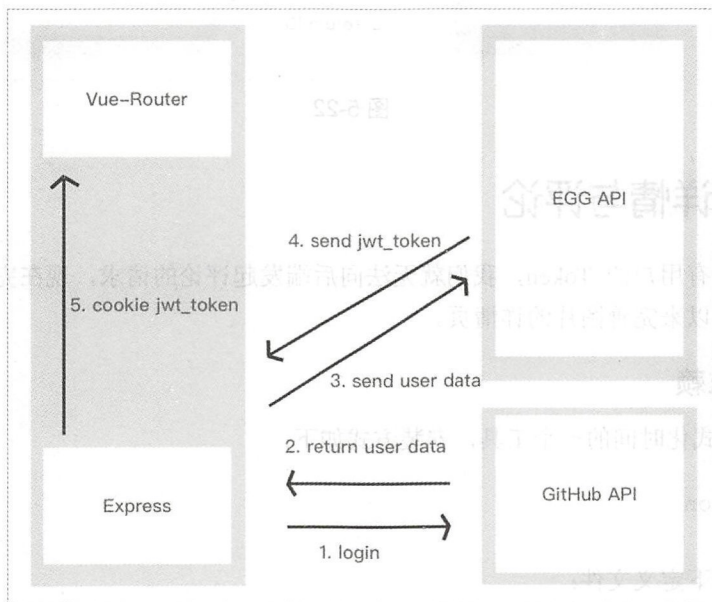


图 5-21

方案二

基于现有的改造，不直接将 `jwt_token` 传到 `query` 里，在 API 服务端先通过 `post` 请求，发送 `jwt_token` 到 SPA 的后端，后端返回一个一次性的 `id`，然后通过 `query` 传递 `id`。用户浏览器的前端获取这个 `id`，再到 SPA 的后端获取 `jwt_token`，如图 5-22 所示。这样的好处就是这个令牌是一次性的，不会出现之前的情况：用户注销了，然后某些怀有恶意的人通过输入网址、浏览器历史而成功实现登录。

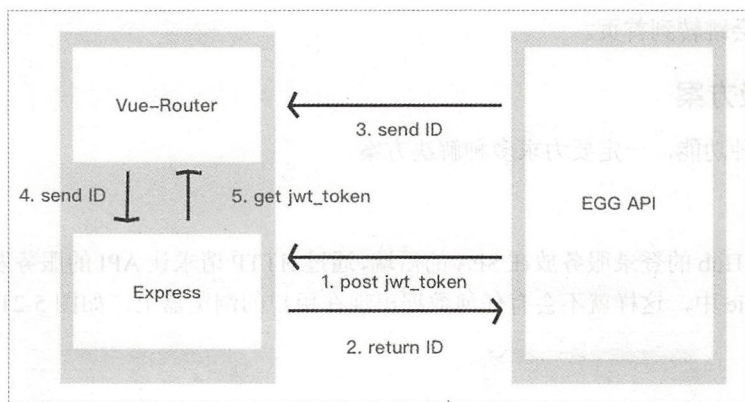


图 5-22

5.8 完善详情与评论

之前由于没有用户的 Token，我们就无法向后端发起评论的请求，现在完成了用户的一些逻辑，我们就可以来完善图片的详情页。

1. 安装依赖

luxon 是格式化时间的一个工具，安装方式如下：

```
npm i luxon
```

同样安装以下定义文件：

```
npm i -D @types/luxon
```

2. 添加全局 filter

在 app.ts 中添加，稍后我们用它来格式化日期。

```
import { Duration, DateTime, Interval } from 'luxon'
Vue.filter('time', function(value) {
  if (!value) return ''
  const start = DateTime.fromISO(value, { locale: 'zh-CN' })
  const i = Interval.fromDateTimes(start, DateTime.local())

  const zhCN = ['年前', '月前', '天前', '小时前', '分钟前', '秒前']
```

```
const values = [
  'years',
  'months',
  'days',
  'hours',
  'minutes',
  'seconds'
].map(v => Math.floor(i.length(v)))
const index = values.findIndex(v => v > 0)
return values[index] + zhCN[index] || '刚刚'
})
```

3. 添加后端逻辑

添加 API

修改 app/api.js:

```
module.exports = ctx => ({
  post: {
    '/comment/img': ctx.comment.image
  },
  get: {
    '/image/:id': ctx.image.show
  }
})
```

创建 comment 控制器

新建 app/controller/comment.js。因为有 JWT 组件，所以从 ctx.state 中可以获取用户的状态。将提交过来的数据创建并返回：

```
'use strict'

const Controller = require('egg').Controller

class CommentController extends Controller {
  async image(ctx) {
    const user = ctx.state.user
    const { image_id, parent_id, content } = ctx.request.body
```

```
const comment = await ctx.model.Imagecomment.create({
  image_id: image_id,
  user_id: user.id,
  parent_id: parent_id,
  content: content
})
ctx.body = comment
}
}

module.exports = CommentController
```

为 image 控制器添加 show 方法

```
async show(ctx) {
  const { Image, Imagecomment, User } = ctx.model
  const { id } = ctx.params
  const where = {
    where: {
      id
    },
    include: [
      {
        model: Imagecomment,
        include: [
          'User',
          { model: Imagecomment, as: 'children', include: ['User'] }
        ]
      },
      {
        model: User
      }
    ]
  }
  const data = await Image.findOne(where)
  ctx.body = data
}
```

4. 添加 SPA API 方法

修改 `src/api/v1/commen.ts`, 导出公用的方法。

```
export function getImage(id) {
  return http
    .get('/image/' + id)
    .then(({ data }) => data)
    .catch(console.error)
}
```

给 `src/api/v1/client/index.ts` 添加推送评论的方法, 不要忘记带上 Token。

```
pushImageComment(token, content, image_id, parent_id) {
  return http.post(
    '/comment/img',
    {
      content,
      image_id,
      parent_id
    },
    {
      headers: {
        Authorization: 'Bearer ' + token
      }
    }
  )
}
```

5. 修改 store 存储

修改 `store/modules/image/index.ts`。

重构后将得到的数据直接放到当前模块的全局中, 而不是多一个 `image`。

`main` 和 `list` 是根据我们之前的约定进行拼接将得到完整的图片路径。

赋值的时候用 `Object.assign` 会好一些, 避免 `state` 更新时, 视图没有更新。

```
import { Module } from 'vuex'
import Vue from 'Vue'
```



```
import { State as RootState } from '../..'

export class State {
  ImageComments = []
  User = {}
  description = ''
  id = 0
  list_url = ''
  main_url = ''
  user_id = 0
  view_id = 0
  updated_at = ''
  created_at = ''
}

const imageFolder = 'http://localhost:7001/public/download/'

const concatImageFolder = imageName =>
  imageFolder + imageName + ',1920,1080.jpg'

const Image: Module<State, RootState> = {
  namespaced: true,
  state: new State(),
  getters: {
    main(state: any) {
      return concatImageFolder(state.main_url)
    },
    list(state: any) {
      if (!state.list_url.length) {
        return []
      }
      return state.list_url.split(',').map(concatImageFolder)
    }
  },
  mutations: {
    SET_IMAGE(state, payload) {
      Object.assign(state, payload)
    },
  },
}
```

```
PUSH_COMMENT(state: any, payload) {  
  state.ImageComments.push(payload)  
}  
},  
actions: {}  
}
```

```
export default Image
```

6. 修改首页视图

利用路由的跳转把 `img` 包裹起来，使用的时候要注意，不要在行内包含块级元素，否则会导致出现 DOM 不匹配的问题。

```
<router-link :to="'/image/'+image.id">  
  <img class="main is-block" v-lazy="'http://127.0.0.1:7001/public/  
download/'+image.main_url+'',1920,1080.jpg'"></img>  
</router-link>
```

7. 修改详情视图

模板

将变量都绑定到视图上，给评论组件绑定了一个 `push` 事件，即单击“提交评论”按钮的时间，它会触发当前页面的 `pushComment` 方法，这属于父子组件通行的内容，不了解的读者可以查阅一下文档，省略了一部分 `photoswipe` 的模板。

```
<div class="container">  
  <div class="columns">  
    <div class="column is-8">  
      <div class="box">  
        <!-- pswp template -->  
        <h3 class="title is-3">{{ image.title }}</h3>  
        <div class="main">  
            
        </div>  
        <div class="list-img">  
            
        </div>  
        <div class="content">
```

```

        {{ image.description }}
      </div>
    </div>
  </div>
<div class="column is-4">
  <div class="box has-text-centered">
    
    <p class="username">{{ image.User.username }}</p>
    <p class="desc small">{{ image.User.desc }}</p>
    <p class="remote" v-if="image.User.receive_remote">
      <i class="fas fa-laptop"></i> 支持远程</p>
    </div>
  </div>
</div>

<app-comment :author="image.User" :comments="image.ImageComments"
type="image" @push="pushComment" />
</div>

```

逻辑

初始化 PhotoSwipe 需要宽和高，不过 URL 中包含了宽和高，我们通过字符串处理，获取它即可。

推送评论，因为本来就是自己评论的，所以 User 设置为自己的用户信息即可，把 children 也初始化一下，否则模板渲染会出错。

```

<script lang="ts">
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State, Getter, Action, Mutation, namespace } from 'vuex-class'
import AppComment from '@C/AppComment.vue'

import PhotoSwipe from 'photoswipe/dist/photoswipe.js'
import PhotoSwipeUI from 'photoswipe/dist/photoswipe-ui-default.js'

const ImageDetailState = namespace('image', State)
const ImageDetailMutation = namespace('image', Mutation)
const ImageDetailGetter = namespace('image', Getter)

```

```
@Component({
  name: 'ImageDetailPage',
  components: {
    AppComment
  }
})
```

```
export default class ImageDetailPage extends Vue {
```

```
  @ImageDetailState(state => state) image: any
```

```
  @ImageDetailGetter('main') main: string
```

```
  @ImageDetailGetter('list') list: string[]
```

```
  @State('user_token') token
```

```
  @State('user_info') user
```

```
  @ImageDetailMutation('PUSH_COMMENT') push_comment
```

```
  galleryOptions = {
```

```
    history: false,
```

```
    focus: false,
```

```
    shareEl: false,
```

```
    showAnimationDuration: 0,
```

```
    hideAnimationDuration: 0
```

```
  }
```

```
  async asyncData({ store, api, route }) {
```

```
    const { id } = route.params
```

```
    const image = await api.getImage(id)
```

```
    store.commit('image/SET_IMAGE', image)
```

```
  }
```

```
  showPhoto() {
```

```
    const item = [this.main].concat(this.list).map(src => {
```

```
      const arr = src.split(',').reverse()
```

```
      const w = parseInt(arr[1]) || 350
```

```
      const h = arr[0].split('.')[0] || 120
```

```
      return {
```

```
        src,
```

```
        w,
```

```
        h
```

```
      }
```

```
    })
```



```

    let gallery = new PhotoSwipe(
      this.$refs.pswp,
      PhotoSwipeUI,
      item,
      this.galleryOptions
    )
    gallery.init()
  }

  async pushComment(this: any, content, parent_id){
    const {data} = await this.$api.pushImageComment(this.token, content,
this.image.id)
    data.User = this.user
    data.childeren = []
    this.push_comment(data)
  }
}

```

8. 修改评论组件

模板

把该替换的部分都替换成变量，然后提交绑定事件。

```

<div class="comment box is-clearfix">
  <h2>评论</h2>
  <ul>
    <li v-for="(comment, index) in comments">
      <div class="comment-header">
        <span class="username">{{ comment.User.username }}</span>
        <span class="publish-date">{{ comment.created_at | time }}</span>
        <span class="comment-footer is-pulled-right">
          <span @click="showComment(index)" v-if="comment.childeren > 0">
            <i class="fas fa-comments small"></i> {{ comment.childeren.
length }}
          </span>
          <span>
            <button>回复</button>

```

```

        </span>
      </span>
    </div>
    <div class="content comment-content">
      <p v-html="comment.content"></p>
    </div>
    <div class="">

  </div>
  <transition name="fade">
    <div class="child-comments" v-show="showIndex == index">
      <div class="comment" v-for="child in comment.children">
        <div class="comment-header">
          <span class="username">{{ child.User.username }}</span>
          <span class="publish-date">{{child.created_at | time}}</span>
        </div>
        <div class="content comment-content">
          <p v-html="child.content"></p>
        </div>
      </div>
      <div v-if="!comment.children.length" class="has-text-centered">
        没有评论</div>
    </div>
  </transition>
</li>
</ul>
<div class="comment-input has-text-centered">
  <button class="button is-small" v-if="!showInput" @click="showInput
= !showInput">我要评论</button>
  <transition name="fade">
    <div v-if="showInput">
      <at v-model="content" :members="members">
        <div class="quill-editor" v-model:content="content" v-quill:
myQuillEditor="editorOption">
          </div>
        </at>
        <button class="button is-small is-pulled-right" @click=
"pushComment">提交</button>

```

```

    </div>
  </transition>
</div>
</div>

```

逻辑

`findAuthorNames` 是一个递归函数，它会查找所有的用户名字。有可能重复，在下面的 `members` 中，直接通过 `new Set` 去重。

当用户单击“提交”按钮的时候，将触发父组件的 `push` 时间，并把用户输入的内容 `content` 当作参数传递过去。

```

import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import At from 'vue-at'

```

```

const findAuthorNames = (comments) => {
  const authorNames = []
  for(let comment of comments) {
    authorNames.push(comment.User.username)
    if (comment.children && comment.children.length) {
      authorNames.concat(findAuthorNames(comment.children))
    }
  }
  return authorNames
}

```

```

@Component({
  name: 'AppComment',
  components: {
    At
  },
  props: ['comments', 'author', 'type']
})

```

```

export default class CommentPage extends Vue {
  showIndex = -1
  content = ''
  editorOption = {
    theme: 'bubble',
    placeholder: '请输入评论'
  }

```

```

    }
    showInput = false
    showComment(index) {
      if (this.showIndex == index) {
        this.showIndex = -1
        return
      }
      this.showIndex = index
    }
    get members(this: any){
      const membersSet = new Set([this.author.username, ...findAuthorNames
(this.comments)])
      return [...membersSet]
    }
    pushComment() {
      this.$emit('push', this.content)
      this.showInput = false
    }
  }
}

```

9. 测试

子评论其实也是类似的功能，只不过提交给后端的是 `parent_id` 而不是 `image_id`，这里就不做重复的东西了。

后端的@功能我们随后再做，目前是可以@当前页作者和评论者的，当然大家也可以通过AJAX 动态地获取输入的相关用户的信息。

5.9 个人中心

本节来完成个人中心页面和邀请码页面，并对代码进行重构，将可复用的代码进行封装。

1. 解决一些 Bug

我们首先解决一些已知存在的问题。项目开发不是一蹴而就的，而是不断完善的过程。

- 首页监听滚动要销毁

修改 `views/index.vue`:

```

mounted(this: any) {
  this.AOSInit()
}

```



```

    if (!this.intersectionObserver) {
      this.intersectionObserver = new IntersectionObserver(entries => {
        if (entries[0].intersectionRatio <= 0) return
        this.loadMore()
        console.log('Loaded new items')
      })
      this.intersectionObserver.observe(document.querySelector('footer.footer'))
    }
  }
  destroyed(this: any) {
    this.intersectionObserver.disconnect()
    delete this.intersectionObserver
  }
}

```

否则会出现莫名其妙的加载，明明是最后一页了，每次还会加载最后一页。

- 当没有 `asyncData` 方法的时候，不会显示 `top` 的载入条

哪怕没有 `asyncData` 方法，我们也要调用一下 `start` 与 `finish` 方法。对于 `beforeResolve` 中的方法同样进行这样的处理。

```

Vue.mixin({
  beforeRouteUpdate(to, from, next) {
    const asyncData = findAsyncDataFunction(this)
    bar.start()
    if (asyncData) {
      asyncData({
        store: this.$store,
        route: to,
        api
      })
      .then(finish(next))
      .catch(fail(next))
    } else {
      Finish(next)()
    }
  }
})

```

2. 重构一些代码

视图名称

保持统一的名称阅读会更好，视图用横杆进行分割，而组件用大驼峰，或者也可以使用统一的大驼峰命名规则，如图 5-23 所示。

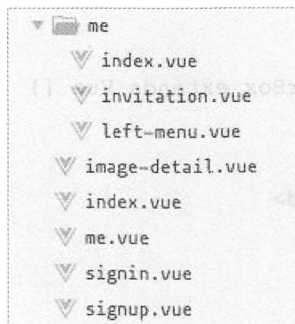


图 5-23

用户信息盒子组件

将图片详情页的用户信息盒子封装成一个组件，供我们在个人中心复用。

```
<div class="column is-4">
  <user-box :user="image.User" />
</div>
```

新建 components/UserBox.vue，内容如下，对 user 进行判断，只有存在才显示，因为目前是测试数据，当有的值不存在时会报错。

```
<template>
  <div class="box has-text-centered" v-if="user">
    
    <p class="username">{{ user.username }}</p>
    <p class="desc small">{{ user.desc }}</p>
    <p class="remote" v-if="user.receive_remote">
      <i class="fas fa-laptop"></i> 支持远程
    </p>
  </div>
</template>

<script lang="ts">
```

```
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
```

```
@Component({
  name: 'UserBox',
  props: ['user']
})
export default class UserBox extends Vue {}
</script>
```

```
<style lang="sass" scoped>
```

```
.box
  box-shadow: none
.avatar
  width: 68px
  height: 68px
  border-radius: 50%
  object-fit: cover
.username
  font-size: 1.5em
  margin: 10px 0 5px
.small
  font-size: .9em
.desc,
.remote
  margin-bottom: 10px
```

```
</style>
```

图片组件

创建 components/TheImage.vue。

对于图片显示，可以把它封装成组件，供个人中心页面来调用。对之前的代码没有做太多的改动，增加了两个 v-if 判断，一个是属性 image，image 用于保证不报错，另一个是属性 showInfo，添加 showInfo 是因为我们想在个人中心不显示用户的信息，因为作者本来就是自己，不必多此一举。

```
<template>
```

```
<div v-if="image">
  <router-link :to="'/image/'+image.id">
    <img class="main is-block" v-lazy="'http://127.0.0.1:7001/public/
download/'+image.main_url+',1920,1080.jpg'"></img>
  </router-link>
  <div class="info">
    <span class="tag">
      <i class="fas fa-eye">{{ image.id }}</i>
    </span>
    <span class="tag">
      <i class="fas fa-comment">{{ image.ImageComments.length }}</i>
    </span>
    <span class="tag">
      <i class="fas fa-heart">22</i>
    </span>
  </div>
  <div class="author" v-if="showInfo">
    <div class="dropdown is-hoverable">
      <div class="dropdown-trigger">
        <a href="#" class="avatar">{{ image.User.username }}</a>
      </div>
      <div class="dropdown-menu">
        <div class="dropdown-content content has-text-centered">
          
          <div class="buttons has-addons is-centered">
            <span class="button is-danger">关注</span>
            <span class="button">私信</span>
          </div>
          <p>我是{{ image.User.username }}, 欢迎关注我, {{ image.User.desc }}</p>
        </div>
      </div>
    </div>
  </div>
</div>
</template>
```



```
<script lang="ts">
export default {
  name: 'TheImage',
  props: ['image', 'showInfo']
}
</script>
```

```
<style lang="sass" scoped>
.column.is-6 .main
  height: auto !important
+touch
  .img-box .main
    height: auto !important
  .img-box
    padding: 12px 0
  .main
    width: 100%
    background-color: #fff
    background-size: cover
    border-color: none
    height: 150px
    padding: 0.4em 0.4em 0
  .info
    background: #fff
    text-align: right
    padding: 0.2em 0
    i::before
      padding-right: .2em
  .tag
    background: #fff
  .author
    margin-top: .6em
  .avatar
    display: flex
    font-size: 1em
    img
      width: 1.5em
```

```
height: 1.5em
margin-right: 10px
border-radius: 50%
</style>
```

图片 DOM 组件

创建 components/Pswp.vue。我们单独把插件所需的 dom 取出来，封装到组件中。

当我们用的时候想获取真实的 DOM 怎么办呢？通过 \$el 实例属性即可获取。

```
<template>
<div class="pswp" tabindex="-1" role="dialog" aria-hidden="true">
  <div class="pswp_bg"></div>
  <div class="pswp_scroll-wrap">

    <div class="pswp_container">
      <div class="pswp_item"></div>
      <div class="pswp_item"></div>
      <div class="pswp_item"></div>
    </div>

    <div class="pswp_ui pswp_ui--hidden">

      <div class="pswp_top-bar">

        <div class="pswp_counter"></div>

        <button class="pswp_button pswp_button--close" title="Close
(Esc)"></button>

        <button class="pswp_button pswp_button--share" title="Share"></button>

        <button class="pswp_button pswp_button--fs" title="Toggle
fullscreen"></button>

        <button class="pswp_button pswp_button--zoom" title="Zoom in/out">
</button>

        <div class="pswp_preloader">
```



```

    <div class="pswp__preloader__icn">
      <div class="pswp__preloader__cut">
        <div class="pswp__preloader__donut"></div>
      </div>
    </div>
  </div>
</div>

<div class="pswp__share-modal pswp__share-modal--hidden pswp__single-
tap">
  <div class="pswp__share-tooltip"></div>
</div>

  <button class="pswp__button pswp__button--arrow--left" title="Previous
(arrow left)">
  </button>

  <button class="pswp__button pswp__button--arrow--right" title="Next
(arrow right)">
  </button>

  <div class="pswp__caption">
    <div class="pswp__caption__center"></div>
  </div>

</div>

</div>
</div>
</template>

<script lang="ts">
export default {
  name: 'Pswp'
}
</script>

```

然后这样使用它：



```

<pswp ref="pswp"/>
let gallery = new PhotoSwipe(
  (this.$refs.pswp as any).$el,
  PhotoSwipeUI,
  item,
  this.galleryOptions
)

```

3. 个人中心首页

个人中心应该包含一些子路由，所以我们会用到子路由的一些知识。

新建 views/me.vue

个人中心左右分割，左边是菜单栏，右边是内容，由于是子路由，所以有一个 router-view 标签做占位符。

```

<template>
  <div class="container">
    <div class="columns">
      <div class="column is-4">
        <left-menu />
      </div>
      <div class="column is-8">
        <router-view />
      </div>
    </div>
  </div>
</template>

```

```

<script lang="ts">
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import LeftMenu from './me/left-menu.vue'

```

```

@Component({
  name: 'MePage',
  components: {
    LeftMenu

```




```

    }
  })
  export default class MePage extends Vue {}
</script>

```

```

<style lang="sass" scoped>
</style>

```

创建 views/me 文件夹

我们在 me 文件夹下放置它的子路由，创建 left-menu.vue，用来存储左边栏的代码。

left-menu.vue

从全局导入用户的信息，渲染出来即可。

```

<template>
  <div>
    <user-box :user="user_info"/>
    <aside class="menu box">
      <p class="menu-label">
        联系我
      </p>
      <ul class="menu-list concat-me" v-if="user_info">
        <li>微博: {{ user_info.weibo }}</li>
        <li>微信: {{ user_info.weixin }}</a></li>
        <li>邮件: {{ user_info.email }}</a></li>
      </ul>
      <p class="menu-label">
        管理菜单
      </p>
      <ul class="menu-list">
        <li><router-link to="/me/invitation">我的邀请码</router-link></li>
      </ul>

    </aside>
  </div>
</template>

```



```

<script lang="ts">
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import UserBox from '@C/UserBox.vue'
import { State } from 'vuex-class'
@Component({
  name: 'LeftMenu',
  components: {
    UserBox
  }
})
export default class LeftMenu extends Vue {
  @State('user_info') user_info
}
</script>

<style lang="sass" scoped>
.box
  box-shadow: none
.concat-me li
  margin: .7rem 0
</style>

```

index.vue

这是默认首页的代码，这里有一个细节要注意，Token 是从 localforage 中获取的，如果服务端渲染，则有可能出现组件已经 mouted 了，全局状态中从 localforage 读取的逻辑还没有完成，所以导致从全局获取 user_token 的是 null。

```

<template>
  <div class="box">
    <div class="columns is-multiline">
      <div v-for="image in images" class="column is-6">
        <the-image :image="image" />
      </div>
    </div>
  </div>
</template>

```



```

<script lang="ts">
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import TheImage from '@C/TheImage.vue'
import * as localforage from 'localforage'

@Component({
  name: 'MeIndex',
  components: {
    TheImage
  }
})
export default class MeIndex extends Vue {
  images = []
  async mounted(this: any) {
    const token = await localforage.getItem('user_token')
    const { data } = await this.$api.getAllImageForMe(token)
    this.images = data
  }
}
</script>

```

4. 个人中心邀请码页

同样从 localforage 获取 Token, 邮件发送笔者就不做了, 后端如何发送邮件之前也阐明了, 前端调用将邀请码发送给用户接口即可。

新建 view/me/invitation.vue

```

<template>
  <div class="box">
    <div class="columns is-multiline">
      <div v-for="i in invitations" class="column is-3">
        <div class="box" :class="{ 'used': i.use_user_id }">
          <p>{{ i.code }}</p>
          <p v-if="i.use_username">已被 {{ i.use_username }} 使用</p>
          <p v-else><a href="#">通过邮件发送</a></p>
        </div>
      </div>
    </div>
  </div>
</template>

```



```

    </div>
  </div>
</template>

<script lang="ts">
import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import * as localforage from 'localforage'

@Component({
  name: 'MeInvitation'
})
export default class MeInvitation extends Vue {
  invitations = []
  async mounted(this: any) {
    const token = await localforage.getItem('user_token')
    const { data } = await this.$api.getAllInvitationForMe(token)
    this.invitations = data
  }
}
</script>

<style lang="sass" scoped>
.box
  box-shadow: none
.is-3 .box
  background: #000
  color: #fff
  text-align: center
  p a
    color: #fff
.is-3 .box.used
  background: #eee
  color: #333
</style>

```

5. 添加路由

me 页面可直接载入，它也不算太大。

```
import Me from '../views/me.vue'
```




```

const MeIndex = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/me/index.vue'], require => {
      resolve(require('../views/me/index.vue'))
    })
  })

const MeInvitation = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/me/invitation.vue'], require => {
      resolve(require('../views/me/invitation.vue'))
    })
  })

```

这里的子路由的 `path` 不要以 “/” 开头，否则 `router` 会认为是根路由。

```

{
  path: '/me',
  component: Me,
  children: [
    {
      path: '',
      component: MeIndex
    },
    {
      path: 'invitation',
      component: MeInvitation
    }
  ]
}

```

6. 完善 API

修改 `api/v1/client/index.ts`

添加如下两个方法，重复地写这个 `headers` 不太好，大家可以尝试自己封装一下。

为什么要放在客户端呢？因为我们不太希望用户的个人中心也渲染出来，只要不在 `store` 中存数据，它就不会渲染出来了。

```

getAllImageForMe(token) {

```



```

return http.get('/user_image', {
  headers: {
    Authorization: 'Bearer ' + token
  }
})
},
getAllInvitationForMe(token) {
  return http.get('/invitation', {
    headers: {
      Authorization: 'Bearer ' + token
    }
  })
}
}

```

修改后端 API

- 在 app/api.js 中添加路由

```

get: {
  '/user_image': ctl.image.forme,
  '/invitation': ctl.invitation.forme
}

```

- 修改 app/controller/image.js

添加 forme 方法，假如 ctx.state 获取不到 user，那说明 JWT 没有正常工作，查看一下 ignore 是否被忽略了。

```

async forme(ctx) {
  const { Image, Imagecomment, User } = ctx.model
  const user = ctx.state.user
  const where = {
    where: {
      user_id: user.id
    },
    include: [
      {
        model: Imagecomment,
        include: [
          'User',
          { model: Imagecomment, as: 'children', include: ['User'] }
        ]
      }
    ]
  }
}

```



```

    ]
  },
  {
    model: User
  }
]
}
ctx.body = await ctx.model.Image.findAll(where)
}

```

- 新建 app/controller/invitation.js

为 Invitation 控制器添加 forme 方法：

```

class InvitationController extends Controller {
  async forme(ctx) {
    ctx.body = await ctx.model.Invitation.findAll({
      where: {
        user_id: ctx.state.user.id
      }
    })
  }
}

module.exports = InvitationController

```

5.10 创建图片

本节将完成图片上传功能，首先要知道这一功能的工作流程，我们要找到一个存储图片的地方，可以是服务提供商，也可以是本地的文件系统，由于做的是图片应用，有大量的图片，所以明智的选择是找一个存储提供商，比如阿里云、腾讯云等都可以，这里笔者选择的是又拍云。

5.10.1 创建又拍云存储

首先进入又拍云官网，创建云存储服务。

创建成功后，你会看到如图 5-24 所示的内容，加速域名就是我们访问的域名，当然默认的这个域名只能用于测试。

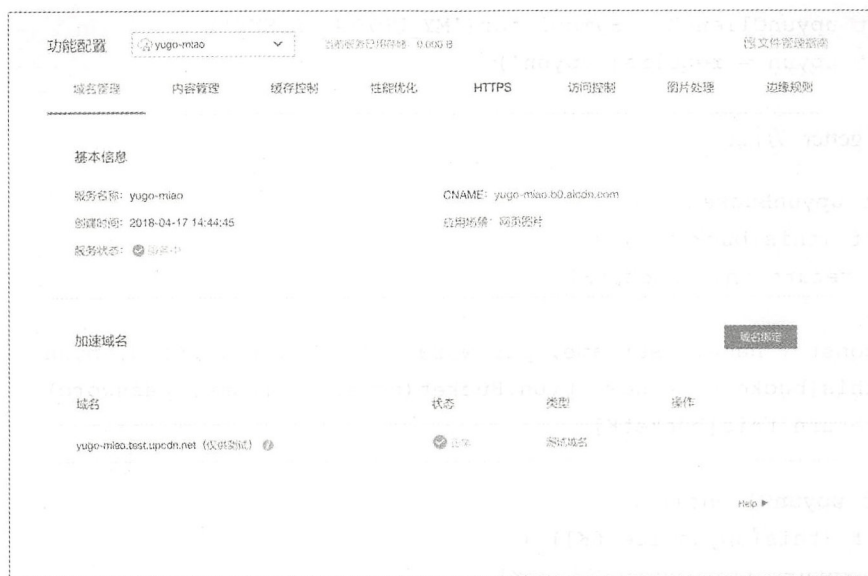


图 5-24

1. 安装 SDK

在前后端都要安装：

```
npm i upyun -S
```

2. 后端配置

在后端的 `config.default.js` 中添加我们刚刚创建服务的配置信息。

```
config.upyun = {
  name: 'yugo-miao',
  username: 'yugo',
  password: 'xxxx'
}
```

在 `ctx` 上创建单例实例，修改 `app/extend/context.js`。

导入对象，这些名字取得不是特别好，大家可以多花点时间思考，取个更符合项目规范的名字。

```
const payK = Symbol.for('MY_PAY')
const bucketK = Symbol.for('MY_BUCKET')
```



```
const upyunClientK = Symbol.for('MY_UPYUN_CLIENT')
const upyun = require('upyun')
```

添加 getter 方法：

```
get upyunBucket() {
  if (this[bucketK]) {
    return this[bucketK]
  }
  const { name, username, password } = this.app.config.upyun
  this[bucketK] = new upyun.Bucket(name, username, password)
  return this[bucketK]
},
get upyunClient() {
  if (this[upyunClientK]) {
    return this[upyunClientK]
  }
  const { name, username, password } = this.app.config.upyun
  this[upyunClientK] = new upyun.Client(
    new upyun.Service(name, username, password)
  )
  return this[upyunClientK]
},
```

5.10.2 添加后端 API

1. 创建路由

将之前的 `image` 修改成 `i_image`，因为之前 JWT 的 `ignore` 是忽略 `image` 的，现在修改成 `i_image`，这样感觉不太符合 RESTful 规范，读者可以写一下更详细的 `ignore` 的规则，或者跟笔者一样，加一个前缀，这里的 `i` 代表 `ignore`。

```
post: {
  '/image': ctl.image.create
},
get: {
  '/i_image': ctl.image.index,
  '/i_image/:id': ctl.image.show,
```

```

'/push_image_token': ctl.image.sign,
'/delete_img_file': ctl.image.delImageFile
}

```

在 config.default.js 的 JWT 配置中修改 ignore:

```

ignore: [
  /\passport/i,
  /\sign/,
  /\i_image/,
  /\email/,
  /\alipay/,
  /\admin/,
  /\.*\.(js|css|map|jpg|png|ico)/
]

```

2. 添加控制器

为了减轻服务器的上传压力，我们直接上传到云提供商的服务器，为了安全，需要签署一个一次性令牌，即 sign 方法。在前端因为跨域的 DELETE 不支持发送删除请求，所以使用了 delImageFile 方法，而 create 则是创建图片的方法。

```

async sign(ctx) {
  ctx.body = upyun.sign.getHeaderSign(
    ctx.upyunBucket,
    ctx.query.method,
    ctx.query.path
  )
}

async delImageFile(ctx) {
  ctx.body = await ctx.upyunClient.deleteFile(ctx.query.path)
}

async create(ctx) {
  const { Image } = ctx.model
  const { title, description, main_url, list_url } = ctx.request.body
  const user_id = ctx.state.user.id
  ctx.body = await Image.create({

```

```
    title,  
    description,  
    main_url,  
    list_url,  
    user_id  
  })  
}
```

3. 存储图片

笔者找了一些国内外图片的 URL 的格式，有的以“users/用户 ID”为前缀，有的以“年/月/日”为前缀，还有的就是随机生成的，没有规律。通常 w/300/h/240 之类都是设置图片大小的，又拍云其实也支持，可能需要多支付一些处理费用。这里使用“user/用户 ID”为前缀的图片，好处是可以扫描出某个用户到底上传了多少文件，假如一些用户恶意上传大文件，占用存储空间，那么我们扫描一下就可以找出来。

5.10.3 前端界面

1. 添加前端 API

修改 api/v1/client/index.ts 添加以下方法，因为 getHeaderToken 要设置 Token，而 bucket、method、path 参数是给回调的，所以要分离出来，可以使用高阶函数的形式。getHeaderToken 代码是从官方 Node.js 的 SDK 示例中复制过来的，如果不知道如何使用，可以看一下代码仓库中的示例。

```
getHeaderToken: user_token => (bucket, method, path) => {  
  return axios  
    .get('http://localhost:7001/api/v1/push_image_token', {  
      headers: {  
        Authorization: 'Bearer ' + user_token  
      },  
      params: {  
        bucket: bucket.bucketName,  
        method,  
        path  
      }  
    })  
}
```

```

    .then(function(response) {
      if (response.status !== 200) {
        console.error('gen header sign faild!')
        return
      }
      return response.data
    })
  },
  deleteFile(token, filePath) {
    return http.get('/delete_img_file?path=' + filePath, {
      headers: {
        Authorization: 'Bearer ' + token
      }
    })
  },
  createImage(token, data) {
    return http.post('/image', data, {
      headers: {
        Authorization: 'Bearer ' + token
      }
    })
  }
}

```

2. 添加路由

导入页面：

```

const MeImageUpload = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/me/image-upload.vue'], require => {
      resolve(require('../views/me/image-upload.vue'))
    })
  })

```

修改子路由：

```

{
  path: '/me',
  component: Me,
  children: [

```



```
{
  path: '/',
  component: MeIndex
},
{
  path: 'invitation',
  component: MeInvitation
},
{
  path: 'new-image',
  component: MeImageUpload
}
]
}
```

3. 创建页面

创建 views/me/image-upload.vue。

4. 添加样式

有一个 hover 的样式，即当鼠标移动到图片上时，会显示一个黑色的遮罩层，上面有一个白色的关闭图标，用来告诉用户，单击该图标可以删除该图片。

```
<style lang="sass" scoped>
.box
  box-shadow: none
#image-container
  width: 100%
  min-height: 200px
  display: flex
  align-items: center
  margin-bottom: 20px
  border: 2px dashed #0a0a0a
  p
    font-size: 1.25rem
    text-align: center
    width: 100%
    color: #0a0a0a
```

```

.list-item
  position: relative
  margin-bottom: 20px
  &:hover::before
    position: absolute
    content: 'X'
    font-size: 2rem
    display: flex
    justify-content: center
    align-items: center
    color: #fff
    background: rgba(0,0,0,.4)
    left: 12px
    right: 12px
    bottom: 35px
    top: 12px
    cursor: pointer
.button
  margin: 1rem 0
  margin-left: auto
.list
  margin-bottom: 2rem
.status
  font-size: .7rem
  text-align: center
  display: block
.success
  color: green
.main-index-select-title
  margin: 1rem 0
</style>

```

5. 添加模板

有一个图片拖曳区域，我们不使用库，笔者也找了一些库，修改起来难度有点大，所以我们自己写，反正也不算太难。下面是一个已经拖曳完成的显示栏，选择主图。然后输入标题与链接，最后是两个按钮，即上传图片按钮与提交表单按钮。

还有一些验证逻辑与提醒用户的逻辑没有添加，比如确保图片都已上传等，读者可以自行添加。

```

<div class="box">
  <div id="image-container" ref="container" @dragover.prevent="noop"
@drop.prevent="imageDrop">
    <p>将图片拖入选项框中</p>
  </div>

  <div class="list">
    <ul class="columns is-multiline">
      <li v-for="(image, index) in images" class="column is-3 list-item"
@click="del(index)">
        
        <span class="status" v-if="!filesStatus[index]">等待上传</span>
        <span class="status success" v-else>完成上传</span>
      </li>
    </ul>
  </div>

  <h2 class="title is-5 main-index-select-title">主图序号</h2>
  <div class="field">
    <div class="control">
      <div class="select is-black is-rounded">
        <select v-model="mainIndex">
          <option v-for="i in files.length" :key="i" :value="i">{{i}}
</option>
        </select>
      </div>
    </div>
  </div>

  <div class="field">
    <div class="control">
      <input class="input is-black" type="text" placeholder="标题"
v-model="title">
    </div>
  </div>

```

```

<div class="field">
  <div class="control">
    <input class="input is-black" type="text" placeholder="简介"
v-model="description">
  </div>
</div>
<button class="button is-black" @click="push">上传图片</button>
<button class="button is-black" @click="submit">提交数据</button>
</div>

```

6. 添加逻辑

要实现拖曳，需要知道几个知识点，当我们将一个图片拖曳进入浏览器的时候，默认用浏览器把它打开，所以第一步就是要阻止这件事的发生。

```

document.cpmdragover = e => {
  e.preventDefault()
}

```

```
@dragover.prevent="noop"
```

上面这两个步骤可以阻止这件事的发生。当拖曳进入 `image-container` 容器中时，会触发 `drop` 事件，`@drop.prevent="imageDrop"` 而在这个事件中，通过 `e.dataTransfer.files` 可以获取上传的文件，然后推入 `files` 中。

`filesStatus` 是图片状态的数组，用来记录转台是否上传成功，`mainIndex` 是主图的序号。`file` 对象不能直接显示，通过 `URL.createObjectURL` 方法可以创建一个本地的临时地址来显示。所以通过 `images.get` 进行转换后，才能正常显示。

此时还需要修改一下图片显示的格式，要重新上传一些测试数据，这里就不改了，大家自行修改即可。通过 `mainIndex` 把主图和列表图切割出来。对于设置图片数组，要通过 `splice` 设置才会响应。

```

import { Vue, Component, Prop } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State } from 'vuex-class'
import * as localforage from 'localforage'
import upyun from 'upyun'

```



```
const bucket = new upyun.Bucket('yugo-miao')
```

```
@Component({
  name: 'MeImageUpload'
})
export default class MeIndex extends Vue {
  files = []
  filesStatus = []
  mainIndex = 0
  description = ''
  title = ''
  get main_url() {
    return this.files[this.mainIndex - 1].name
  }
  get list_url() {
    let filenames = this.files.map(f => f.name)
    filenames.splice(this.mainIndex - 1, 1)
    return filenames.join(',')
  }
  noop() {}
  @State('user_token') user_token
  @State('user_info') user
  get images() {
    return this.files.map(v => URL.createObjectURL(v))
  }
  imageDrop(e) {
    const fileList = e.dataTransfer.files
    Array.from(fileList).forEach(file => this.files.push(file))
  }

  async del(this: any, index) {
    const _del = () => {
      this.files.splice(index, 1)
      this.filesStatus.splice(index, 1)
    }
    if (!this.filesStatus[index]) {
      _del()
      return
    }

    const { data } = await this.$api.deleteFile(
```

```

        this.user_token,
        `/user/${this.user.id}/image/${this.files[index].name}`
    )
    if (data) {
        _del()
    }
    console.log(data)
}

async mounted(this: any) {
    const token = await localforage.getItem('user_token')
    if (!token) {
        this.$router.push('/signin')
        return
    }
    document.ondragover = e => {
        e.preventDefault()
    }
}

get client(this: any) {
    if (this['_client']) {
        return this['_client']
    }
    this['_client'] = new upyun.Client(
        bucket,
        this.$api.getHeaderToken(this.user_token)
    )
    return this['_client']
}

async push() {
    for (let index in this.files) {
        const file = this.files[index]
        const ok = await this.client.putFile(
            `/user/${this.user.id}/image/${file.name}`,
            file
        )
        this.filesStatus.splice(index as any, 1, ok)
    }
}

```

```
async submit(this: any) {
  if (this.filesStatus.some(f => f == false)) {
    return
  }
  const image = await this.$api.createImage(this.user_token, {
    main_url: this.main_url,
    list_url: this.list_url,
    description: this.description,
    title: this.title
  })
  console.log(image)
}
```

5.10.4 测试

1. 将图片拖入选项框中

2. 删除图片

移动到图片上面可以看到一个删除按钮，单击“×”即可删除图片。

3. 单击“上传图片”按钮

状态会显示“完成上传”。

4. 查看服务中是否上传完成

下载并打开 FileZilla，地址填 v0.ftp.upyun.com，用户名填“操作员/存储桶名称”，密码是操作员的密码，单击“连接”按钮即可，效果如图 5-25 所示。

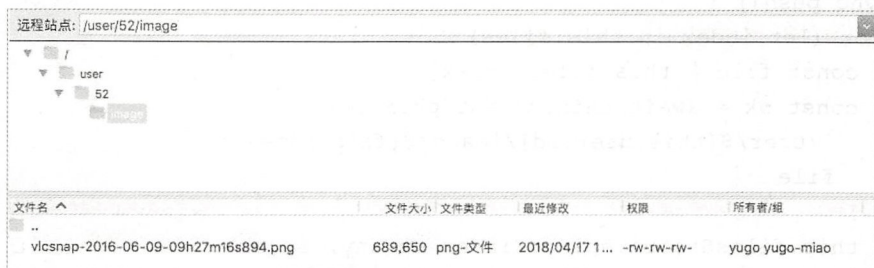


图 5-25

可以看到刚刚上传的文件，不过只有一个，因为两个文件的名称都是一样的，可以用 `uuid` 生成唯一的图片名。

5. 提交数据

提交数据并打印返回值，即可看到结果如图 5-26 所示。可能之前后端设置的地址太短了，所以会报错，读者可以把表字段改长一些。

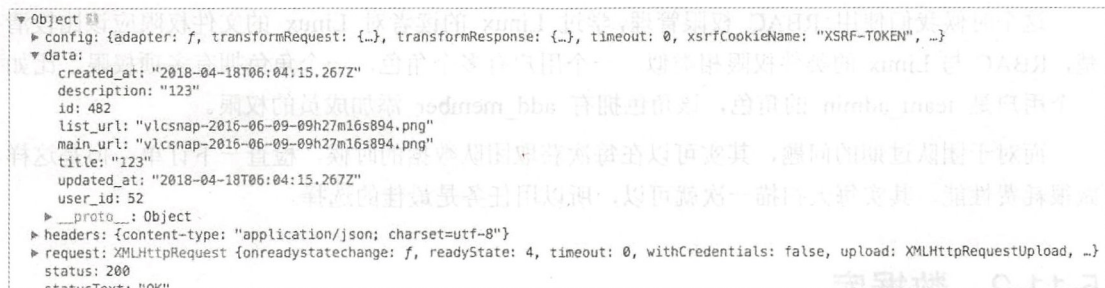


图 5-26

5.11 团队

这是前端部分的最后一节了，剩余的一些功能基本都类似，所以笔者就不一一实现。关于团队的功能其实牵扯的表也比较多，流程也稍显复杂，所以要先思考以怎样的流程、完成哪些任务，才能实现这个功能。

这里笔者简单地做了一个思维导图，如图 5-27 所示。其实思考得也不是很全面，不过可以先想一个大致的提纲，在具体做的时候再进行细化。

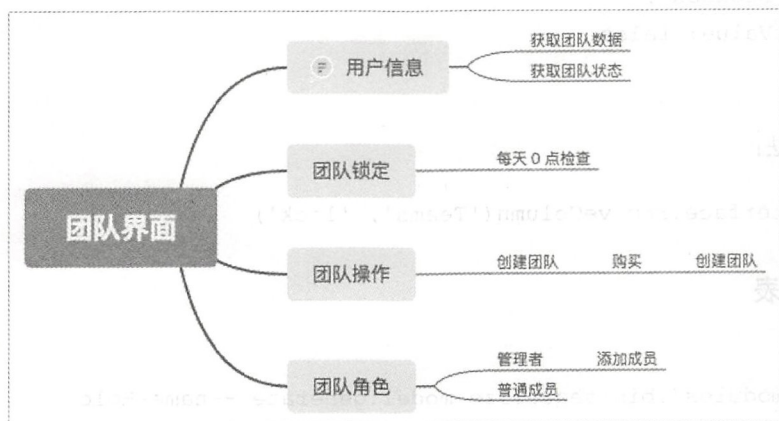


图 5-27

5.11.1 功能是如何工作的

用户支付成功后，可以创建团队、添加成员，然后成员每发布一些新的东西，都会出现在团队状态中。因为团队是按年付费的，超时应锁定，所以给 Team 添加一个 lock 字段。团队中除了创建者可以添加新的成员，还可以把某些用户设置为团队管理员，让其也可以添加用户，这个时候就需要有权限管理的机制。

这个时候我们使用 RBAC 权限管理，学过 Linux 的读者对 Linux 的文件权限应该比较清楚，RBAC 与 Linux 的类件权限相类似。一个用户有多个角色，一个角色拥有多项权限。比如一个用户是 team_admin 的角色，该角色拥有 add_member 添加成员的权限。

而对于团队过期的问题，其实可以在每次获取团队数据的时候，检查一下订单，但是这样做很耗费性能，其实每天扫描一次就可以，所以用任务是最佳的选择。

5.11.2 数据库

1. 给 Team 添加 lock 字段

命令：

```
./node_modules/.bin/sequelize migration:create --name team_add_lock_field
```

up 方法：

```
queryInterface.addColumn('Teams', 'lock', {  
  type: 'boolean',  
  defaultValue: false  
})
```

down 方法：

```
queryInterface.removeColumn('Teams', 'lock')
```

2. 角色表

命令：

```
./node_modules/.bin/sequelize model:generate --name Role
```

Migration:

```

queryInterface.createTable('Roles', {
  id: {
    allowNull: false,
    autoIncrement: true,
    primaryKey: true,
    type: Sequelize.INTEGER
  },
  name: {
    type: Sequelize.STRING
  },
  meta: {
    type: Sequelize.TEXT
  }
})

```

Model:

```

'use strict'

module.exports = ({ model: sequelize, Sequelize: DataTypes }) => {
  const Role = sequelize.define(
    'Role',
    {
      name: DataTypes.STRING,
      meta: DataTypes.TEXT
    },
    {
      timestamps: false
    }
  )

  return Role
}

```

3. 权限表

命令：

```
./node_modules/.bin/sequelize model:generate --name Permission
```

Migration:

```
queryInterface.createTable('Permissions', {
  id: {
    allowNull: false,
    autoIncrement: true,
    primaryKey: true,
    type: Sequelize.INTEGER
  },
  name: {
    type: Sequelize.STRING
  },
  meta: {
    type: Sequelize.TEXT
  }
})
```

Model:

```
module.exports = ({ model: sequelize, DataTypes }) => {
  const Permission = sequelize.define(
    'Permission',
    {
      name: DataTypes.STRING,
      meta: DataTypes.TEXT
    },
    {
      timestamps: false
    }
  )

  return Permission
}
```

4. 用户角色关联表

命令:

```
./node_modules/.bin/sequelize model:generate --name UserRole
```

Migration:

```
queryInterface.createTable('UserRoles', {
  id: {
    allowNull: false,
    autoIncrement: true,
    primaryKey: true,
    type: Sequelize.INTEGER
  },
  user_id: {
    allowNull: false,
    type: Sequelize.INTEGER
  },
  role_id: {
    allowNull: false,
    type: Sequelize.INTEGER
  }
})
```

Model:

```
'use strict'
```

```
module.exports = app => {
  const { model: sequelize, Sequelize: DataTypes } = app
  const UserRole = sequelize.define(
    'UserRole',
    {
      user_id: DataTypes.INTEGER,
      role_id: DataTypes.INTEGER
    },
    {
      timestamps: false
    }
  )
```



```

    }
  )

  UserRole.associate = function() {
    app.model.Role.belongsToMany(app.model.User, {
      through: {
        model: this,
        unique: false
      },
      foreignKey: 'role_id'
    })

    app.model.User.belongsToMany(app.model.Role, {
      through: {
        model: this,
        unique: false
      },
      foreignKey: 'user_id'
    })
  }

  return UserRole
}

```

5. 角色权限关联表

命令：

```
./node_modules/.bin/sequelize model:generate --name RolePermission
```

Migration:

```

queryInterface.createTable('RolePermissions', {
  id: {
    allowNull: false,
    autoIncrement: true,
    primaryKey: true,
    type: Sequelize.INTEGER
  },

```

```

    role_id: {
      type: Sequelize.INTEGER
    },
    permisstion_id: {
      type: Sequelize.INTEGER
    }
  })
  /node_modules/.bin/sequelize db:Migration

```

Model:

```

'use strict'

module.exports = app => {
  const { model: sequelize, Sequelize: DataTypes } = app
  const RolePermission = sequelize.define(
    'RolePermission',
    {
      role_id: DataTypes.INTEGER,
      permisstion_id: DataTypes.INTEGER
    },
    {
      timestamps: false
    }
  )
  RolePermission.associate = function() {
    app.model.Role.belongsToMany(app.model.Permisstion, {
      through: {
        model: this,
        unique: false
      },
      foreignKey: 'role_id'
    })

    app.model.Permisstion.belongsToMany(app.model.Role, {
      through: {
        model: this,
        unique: false
      }
    })
  }
}

```



```

    },
    foreignKey: 'permisstion_id'
  })
}
return RolePermission
}

```

6. 修正 Team 的关联关系

修改 app/model/team.js 的 associate 方法：

```

app.model.User.hasOne(Team, {
  foreignKey: 'creator_id'
})
app.model.Team.belongsTo(app.model.User, {
  foreignKey: 'creator_id',
  as: 'creator'
})

Team.hasMany(app.model.User)
app.model.User.belongsTo(Team)

```

5.11.3 后端

1. 添加后端路由

修改 app/api.js 的路由配置项：

- /team/member，给团队添加成员的接口；
- /team/create，创建团队的接口；
- /search_user，用户查询接口，因为我们要给团队添加用户，所以要有一个搜索用户的接口；
- /team/:id，查询团队的数据；
- /check/team，检测用户是否可以创建团队。

```

{
  post: {

```



```

    '/team/member': ctl.team.member,
    '/team/create': ctl.team.create
  },
  get: {
    '/team/:id': ctl.team.show,
    '/search_user': ctl.team.search,
    '/check/team': ctl.team.checkCanCreat
  }
}

```

2. 添加 Team 控制器

创建 app/controller/team.js:

- show, 用来显示团队的信息, 表链接有些多, 由于成员需要显示其角色, 所以我们要连接 Role 表, 而且这些数据不应该返回给前端, 判断完一定要将这些权限的数据进行删除;
- checkPermission, 检测是否拥有管理员角色;
- member, 添加用户到团队中;
- checkCanCreat, 判断是否可以创建团队, 要想创建必须满足两个条件, 一是没有任何团队, 二是有一个最近一年内付费的有效订单;
- create, 创建团队方法, 同时别忘记要给用户加上 team_admin 角色, 成功后要重新签署用户 Token, 因为数据更新了。

```
'use strict'
```

```
const Controller = require('egg').Controller
```

```

class TeamController extends Controller {
  async show(ctx) {
    const team_id = ctx.params.id
    const { Team, Teamstatus, User, Role, Permisstion } = ctx.model
    console.log(ctx.model)
    let data = await Team.findOne({
      where: {
        id: team_id
      },

```




```

include: [
  {
    model: Teamstatus,
    include: [{ model: User }]
  },
  {
    model: User,
    include: [
      {
        model: Role,
        include: [{ model: Permission }]
      }
    ]
  }
]
})
data = data.toJSON()
for (const index in data.Users) {
  data.Users[index].rolename = data.Users[index].Roles.some(
    role => role.name === 'team_member'
  )
  ? '会员'
  : '管理员'
  delete data.Users[index].Roles
}
ctx.body = data
}

async checkPermission(ctx, user_id) {
  const user = ctx.model.User.findOne({
    where: { id: user_id },
    include: [
      {
        model: ctx.model.Role
      }
    ]
  })
  const isAdmin = user.Roles.some(role => role.name === 'team_admin')

```



```

    ctx.assert(isAdmin, 403)
  }

  /**
   * body: user_id、team_id
   */
  async member(ctx) {
    const { user_id, team_id } = ctx.request.body
    const { Team, User } = ctx.model
    const current_user_id = ctx.state.user.id
    const team = await Team.findById(team_id)
    this.checkPermission(ctx, current_user_id)
    const member = await User.findById(user_id)
    member.team_id = team.id
    await member.save()
    ctx.body = 'success'
  }

  async search(ctx) {
    const k = ctx.query.k
    const { User } = ctx.model
    const users = await User.findAll({
      where: {
        username: {
          [ctx.app.Sequelize.Op.like]: '%' + k + '%'
        }
      },
      limit: 6
    })
    ctx.body = users
  }

  async checkCanCreat(ctx) {
    const user_id = ctx.state.user.id
    const user = await ctx.model.User.findById(user_id)
    ctx.body = false
    if (!user.team_id) {
      const onYearAgo = new Date()

```



```
onYearAgo.setFullYear(onYearAgo.getFullYear() - 1)
const orders = await ctx.model.Order.findAll({
  where: {
    state: 1,
    created_at: {
      [this.ctx.app.Sequelize.Op.lt]: new Date(),
      [this.ctx.app.Sequelize.Op.gt]: onYearAgo
    }
  },
  order: ['created_at']
})
if (orders.length > 0) {
  ctx.body = true
}
}
}

async create(ctx) {
  this.checkCanCreat(ctx)
  if (ctx.body === false) {
    ctx.throw(403)
  }
  const { Team, User, Userrole, Role } = ctx.model
  console.log(ctx.model)
  const user = await User.findById(ctx.state.user.id)
  const { name, profile } = ctx.request.body
  const team = await Team.create({
    name,
    profile,
    creator_id: user.id
  })
  const adminRole = await Role.findOne({
    where: {
      name: 'team_admin'
    }
  })
  user.team_id = team.id
  await user.save()
```



```
await Userrole.create({
  role_id: adminRole.id,
  user_id: user.id
})
const raw_user = require('ramda').omit(
  ['password', 'created_at', 'updated_at'],
  user.toJSON()
)
ctx.body = await ctx.sign_token(raw_user)
}
}

module.exports = TeamController
```

3. 团队动态

修改 `app/controller/image.js` 控制器的 `create` 方法，添加以下内容，假如用户有所属的团队，则需要给这个团队添加以下团队的动态。

```
if (ctx.state.user.team_id) {
  await Teamstatus.create({
    user_id,
    title,
    team_id: ctx.state.user.team_id,
    image_url: main_url,
    type_link: ctx.body.id,
    type: 'image'
  })
}
```

4. 检测团队是否过期

创建 `app/schedule/check_team_status.js` 文件。

- `schedule`，返回的值是对任何计划的配置，表示一天一次，在 `worker` 进程上执行
- `subscribe`，其中的代码就是我们检测的逻辑，寻找团队创建者

```
'use strict'
```

```
const Subscription = require('egg').Subscription
```




```
class TeamStatusChecker extends Subscription {
  static get schedule() {
    return {
      interval: '5s',
      type: 'worker'
    }
  }

  async subscribe() {
    const onYearAgo = new Date()
    onYearAgo.setFullYear(onYearAgo.getFullYear() - 1)
    const teams = await this.ctx.model.Team.findAll({
      include: [
        {
          model: this.ctx.model.User,
          as: 'creator'
        }
      ]
    })
    for (const team of teams) {
      const user_id = team.creator.id
      const orders = await this.ctx.model.Order.findAll({
        where: {
          user_id,
          state: 1,
          created_at: {
            [this.ctx.app.Sequelize.Op.lt]: new Date(),
            [this.ctx.app.Sequelize.Op.gt]: onYearAgo
          }
        },
        order: ['created_at']
      })
      if (orders.length === 0 && team.lock === false) {
        team.lock = true
        await team.save()
        console.log(team.name + ' LOCK')
```



```
    } else {
      team.lock = false
      await team.save()
      console.log(team.name + ' OPEN')
    }
  }
}
```

```
module.exports = TeamStatusChecker
```

5. 修改支付成功重定向地址

修改 `app/controller/pay.js` 中的 `success` 方法:

```
ctx.unsafeRedirect(ctx.app.config.frontURL + '/me/team')
```

5.11.4 前端

1. 添加前端 API

添加如下几个方法:

```
getTeam(token, id) {
  return http.get('/team/' + id, {
    headers: {
      Authorization: 'Bearer ' + token
    }
  })
},
search(token, key) {
  return http.get('/search_user?k=' + key, {
    headers: {
      Authorization: 'Bearer ' + token
    }
  })
},
addMember(token, user_id, team_id) {
```



```
return http.post(
  '/team/member',
  {
    user_id,
    team_id
  },
  {
    headers: {
      Authorization: 'Bearer ' + token
    }
  }
),
checkCanCreatTeam(token) {
  return http.get('/check/team', {
    headers: {
      Authorization: 'Bearer ' + token
    }
  })
},
createTeam(token, data) {
  return http.post('/team/create', data, {
    headers: {
      Authorization: 'Bearer ' + token
    }
  })
}
```

2. 添加路由

导入页面：

```
const MeTeam = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/me/team/index.vue'], require => {
      resolve(require('../views/me/team/index.vue'))
    })
  })
```

```
const MeTeamCreate = () =>
  new Promise((resolve, reject) => {
    require.ensure(['../views/me/team/create.vue'], require => {
      resolve(require('../views/me/team/create.vue'))
    })
  })
})
```

路由配置:

```
{
  path: 'team',
  component: MeTeam
},
{
  path: 'team/create',
  component: MeTeamCreate
}
```

3. 团队页

创建 `views/me/team/index.vue` 页面, 当没有团队的时候, 显示团队创建按钮; 当锁定的时候显示续费按钮; 当有团队的时候, 显示团队内容。

还有一部分验证逻辑没有添加, 比如用户的团队可以被覆盖的后端验证逻辑, 以及添加成员按钮显示的前端验证。

`getData` 用于获取团队数据的封装, 分别在路由和监听的 `state` 中获取, 还是由于之前的问题, 有可能 `Token` 还没从 `LocalStorage` 中获取, 当没有获取 `Token` 的时候发起请求就会报错, 但是又要保证它发起了请求, 所以我们写了两个。当没有 `Token` 的时候会 `return`, 所以其实只会执行一次。

`getMember` 中加了一个函数去抖动, 即延迟 400 毫秒执行, 当频繁触发的时候, 只保证最后一次有效。

```
<template>
  <div>
    <div class="notification has-text-centered" v-if="msg">
      <button class="delete" @click="msg = ''></button>
      {{ msg }}
    </div>
```



```

<div class="no-permission" v-if="!hasTeam">
  <div class="notification">
    <p>当前，您没有加入任何团队，或许您可以创建一个。</p>
    <router-link class="button is-black" to="/me/team/create">创建团队
  </router-link>
  </div>
</div>
<div v-if="lock">
  <p>{{team.name}} 已被锁定，请续费</p>
  <router-link class="button is-black" to="/me/team/create">续费
</router-link>
</div>
<div class="team-block" v-if="team">
  <h3 class="title is-3">
    团队成员
    <button class="button is-black is-pulled-right" @click="show_add_
panel=true">添加成员</button>
  </h3>
  <div class="columns">
    <div class="column is-3" v-for="user in team.Users">
      <div class="media">
        <div class="media-left">
          <p class="image is-64x64">
            
          </p>
        </div>
        <div class="media-content">
          <h2 class="title is-4">{{user.username}}</h2>
          <p class="subtitle">{{ user.rolename}}</p>
        </div>
      </div>
    </div>
  </div>
  <div class="columns">
    <div class="column is-3" v-for="status in team.TeamStatuses">
      <p class="time-ago">{{ status.created_at | time }}</p>
    </div>
  </div>
</div>
<h3 class="title is-3">团队动态</h3>
<div class="status">
  <div class="status-item" v-for="status in team.TeamStatuses">
    <p class="time-ago">{{ status.created_at | time }}</p>
  </div>
</div>

```

了作品

```
<router-link to="/">{{ status.User.username }}</router-link> 发布

<router-link to="/">{{ status.title }}</router-link>

</div>
</div>
</div>

<div class="add_member_panel" v-if="show_add_panel">
  <div class="field">
    <label class="label">搜索用户
      <span class="close is-pulled-right is-small" @click="show_add_panel=
false">关闭</span>
    </label>

    <div class="control">
      <input class="input is-black" type="text" placeholder="请输入用户
名" v-model="search_key">
    </div>
  </div>

  <ul>
    <li v-for="user in members" class="is-clearfix member-item">
      {{user.username}}
      <button class="button is-black is-pulled-right is-small" @click=
"addMember(user.id)">添加为成员</button>
    </li>
  </ul>
  <div v-if="!members.length">暂无用户</div>
</div>
</div>
</template>

<script lang="ts">
import { Vue, Component, Prop, Watch } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State } from 'vuex-class'
```

```
import * as localforage from 'localforage'

let timer = null

@Component({
  name: 'MeTeam',
  beforeRouteEnter(this: any, to, from, next) {
    next((vm: any) => {
      if (vm.team == false) {
        vm.getData()
      }
      next()
    })
  }
})

export default class MeIndex extends Vue {
  hasTeam = false
  lock = false
  team = false
  search_key = ''
  members = []
  show_add_panel = false
  msg = ''
  @State('user_info') info
  @State('user_token') token

  @Watch('info')
  async infoChange(this: any, newValue, oldValue) {
    this.getData()
  }

  @Watch('search_key')
  async getMember(this: any, newValue, oldValue) {
    ~((() => {
      if (timer) {
        clearTimeout(timer)
      }
      timer = setTimeout(async () => {
```

```
const { data } = await this.$api.search(this.token, this.search_key)
console.log(data)
this.members = data
}, 300)
)) ()
}

async addMember(this: any, user_id) {
  const { data } = await this.$api.addMember(
    this.token,
    user_id,
    this.team.id
  )
  this.msg = '添加成员成功'
  this.show_add_panel = false
}

async getData(this: any) {
  console.log('GET DATA')
  if (!this.info) {
    return
  }
  const { data } = await this.$api.getTeam(this.token, this.info.team_id)
  if (data) {
    this.hasTeam = true
  }
  if (data.lock) {
    this.lock = true
  }
  this.team = data
}
async mounted(this: any) {}
}

</script>

<style lang="sass" scoped>
.member-item
  margin: 10px 0
```



```

.close
    cursor: pointer
.status-item
    margin: 1rem 0
.add_member_panel
    position: absolute
    width: 50%
    height: 400px
    padding: 20px
    top: 20%
    background: #fff
</style>

```

4. 创建页

新建 `views/me/team/create.vue` 页面

创建页比之前的页面简单得多，这是由于成功后修改了 `user` 的数据，导致 `watch` 会再次执行，所以多加了一个 `checked` 变量来标识是否已检测。

```

<template>
  <div>
    <div class="field">
      <label class="label">团队名字</label>
      <div class="control">
        <input class="input" type="text" placeholder="团队名字" v-model=
"name">
      </div>
    </div>
    <div class="field">
      <label class="label">团队简介</label>
      <div class="control">
        <input class="input" type="text" placeholder="团队简介" v-model=
"profile">
      </div>
    </div>
    <button class="button is-black" @click="submit">创建</button>
  </div>
</template>

```

```
<script lang="ts">
import { Vue, Component, Prop, Watch } from 'vue-property-decorator'
import { ComponentOptions } from 'vue'
import { State } from 'vuex-class'
import * as localforage from 'localforage'

@Component({
  name: 'MeTeamCreate',
  beforeRouteEnter(this: any, to, from, next) {
    next((vm: any) => {
      vm.check(vm.token)
      next()
    })
  }
})

export default class MeTeamCreate extends Vue {
  @State('user_token') token
  @State('user_info') user
  name = ''
  profile = ''
  checked = false
  @Watch('user')
  async check(this: any) {
    console.log(this)
    if (!this.token || this.checked) {
      return
    }
    try {
      const { data } = await this.$api.checkCanCreatTeam(this.token)
      if (data == false) {
        window.location.href = 'http://localhost:7001/alipay/pay/10'
      }
    } catch (e) {
      window.location.href = 'http://localhost:7001/alipay/pay/10'
    }
    this.checked = true
  }
}
```

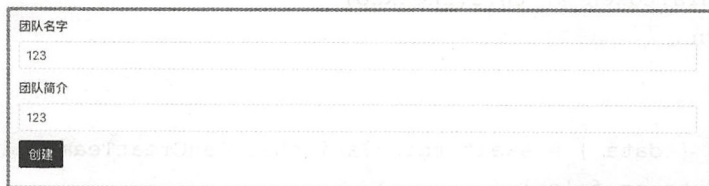
```
async submit(this: any) {  
  const { data } = await this.$api.createTeam(this.token, {  
    name: this.name,  
    profile: this.profile  
  })  
  if (data) {  
    await localforage.setItem('user_token', data)  
    this.$store.commit('USER_TOKEN', data)  
    const { data: { user } } = await this.$api.getUser(data)  
    this.$store.commit('USER_INFO', user)  
    this.$router.push('/me/team')  
  }  
}  
}  
</script>
```

5.11.5 测试

1. 创建团队

首先删除本用户的 `user_id` 和相关的 `Team`，以及 `Orders` 里有效的订单。然后会显示“创建团队”按钮。

单击“创建团队”按钮之后，来到“创建”页面，输入数据，如图 5-28 所示。



团队名字
123
团队简介
123
创建

图 5-28

单击“创建”按钮之后，回到了团队首页。

2. 添加成员

单击“添加成员”按钮，输入 A，搜索 A 开头的用户，单击“添加为成员”按钮，如图 5-29 所示。

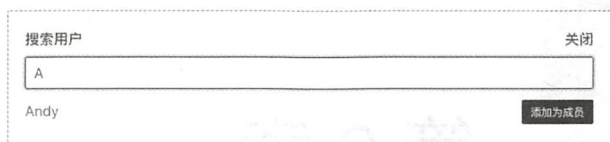


图 5-29

会看到提示消息，因为没有重新获取数据，所以没有更新，刷新一下即可。

可以看到 Andy 现在是成员了，如图 5-30 所示。



图 5-30

6 chapter

第 6 章 部署与运维

6.1 认识 Docker

Docker 是基于 Google 的 Go 语言开发的一种容器技术。利用 Linux 内核技术 Namespace 和 Cgroup 实现隔离与分配资源，进而实现一种可随时启动和销毁容器的能力。

由于本书面向的是 Web 开发人员，而不是专业的 DevOps，本章只会讲解一些 Docker 的常用概念，以及如何使用它，以帮助你更快地实现部署，而不会对所有概念都进行阐述。

6.1.1 解决了什么问题

1. 频繁搭建环境

相信读者一定听说过 Linux 运维工程师这个职业，这个职业的职能就是搭建 Linux 的运行环境与维护系统的运行。在企业中，后台服务器其实是非常多的，毕竟要支撑起庞大的业务量。那么逐一安装每一台系统环境势必会是一个吃力不讨好的事，最初的时候 Linux 工程师会维护自己的一个 Bash 脚本库，每安装一台机器就会运行一次写好的脚本。但是速度还是不够快，需要找到一种方式把脚本推送到安装服务器上。后来出现了用编程语言来控制安装服务环境，比如 Ansible、SaltStack 等工具，解决了运行环境的问题。

2. 环境不一致

但是此时又面临另外的问题，开发人员的环境和运维人员的环境不一致，就会出现在线下可以成功运行、但是在线上总是频频出错的情况。这是个非常棘手的问题，因为开发人员原生的 Windows 环境不可能跟线上的 Linux 保持一致。于是就有人想到了使用虚拟化 VirtualBox，VirtualBox 是 Oracle 开源的虚拟化软件，是基于计算机硬件设备的虚拟化，通过 VirtualBox 可以让开发人员在 Windows 上虚拟化一个 Linux 系统，但是这种方式还是需要开发人员安装虚拟化出来的操作系统的环境。为了解决这一问题出现了一个叫 Vagrant 的软件，VirtualBox 提供了命令行接口，Vagrant 定义了一套自己访问 VirtualBox 的接口，Vagrant 可以通过一个 Vagrantfile 文件定义如何快速启动一个别人烧录好的镜像，把源代码通过网络文件系统同步到虚拟化环境中，比如一个烧录好了 Nginx 环境的镜像。

3. 二次虚拟化

既然可以在本地通过 Vagrant 启动一个虚拟化的环境来运行应用，那么可不可以在线上也使用 Vagrant 呢？答案是不行的，为什么呢？硬件应该是不支持二次虚拟化的，笔者尝试在线上安装是不成功的，但是有的人经过一些比较 Hack 的技巧后用 VMware 尝试成功了。我们购买的云服务器其实就是经过虚拟化的了，并不是实际的服务器，当然也有实际的服务器，但是价格会高昂很多，远不如云服务器划算。

通常我们购买云服务器的平台叫作公有云计算平台，它拥有数据中心和机房、网路拓扑、灾备中心等非常多的基础设施，所以也叫 IAAS（基础设施即服务）。IAAS 中都是非常多的刀片式服务器，刀片式服务器的性能都不低，不可能出现 1 核 1GB 内存的刀片式服务器，但是为什么那些公有云又有 1 核 1GB 的内存套餐呢？其实这就是用了云计算，云计算就是把实际的硬件设备都整合起来，放到云上，当你需要计算资源的时候，无论是计算机、存储、还是网络都可以向云进行申请，一般公有云还会有计费模块。而公司也可以自己搭建私有云，基于 Python 开发的 OpenStack 就是搭建云计算平台的一个开源软件。

硬件无法进行二次虚拟化，那么有没有基于软件的虚拟化呢？于是就出现了 Docker，Docker 同样可以让你写一份 Dockerfile 文件，运行在任何地方。

6.1.2 使用 Docker 的流程

这里我们不说如何使用 Docker，而是以全局观来看一下 Docker 是如何帮助我们进行开发与部署的。

通常在开发的时候，我们首先会写一个或者多个 Dockerfile 文件，文件中定义了如何启动一个容器，一个 Dockerfile 对应一个容器，Dockerfile 描述了容器中安装哪些软件，比如 Redis、

MySQL、Nginx 等。

```
FROM node:slim

ENV DEBIAN_FRONTEND noninteractive
ENV NODE_ENV development

WORKDIR /app

RUN apt-get update && apt-get install -y build-essential python git && \
    apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* && \
    npm i -g typescript supervisor && \
    npm cache clean

CMD bash
```

Dockerfile 文件会有很多指令，后面章节再进行详细的说明，我们所创建的运行 Node.js 的容器会跟上面的类似，现在看一下 Dockerfile，Dockerfile 中大多是 Linux 命令。当我们写好了 Dockerfile 之后，按照这个文件的配置烧录一个可以运行的镜像，这里理解成制作一个 USB 启动盘即可了。当制作完这个镜像之后，就可以运行这个镜像，运行成功之后就变成了容器。一般我们自定义的镜像都是基于基础镜像构建的，比如 Ubuntu、CentOS 等。开发完成后通过 docker commit 命令将容器制作成镜像，然后上传到自己的私有仓库中，在线上服务器拉取这个镜像运行即可。

但是，笔者建议把这些提供服务的软件放在不同的容器里，容器之间要进行通信，则需要通过命令行进行连接。

运行一个 Redis 服务，只需要一个简单的命令：

```
docker run --name some-redis -d redis
```

连接一个 Redis 服务，通过 redis-cli 在命令行界面操作它：

```
docker run -it --link some-redis:redis --rm redis redis-cli -h redis -p 6379
```

这时要写太多的 Dockerfile，还要逐个去启动，手动去连接，效率很低。所以就有了 docker-compose.yml，通过配置这个文件，可以同时启动多个容器，并定义它们之间的存储和网络关系，然后让这些容器提供服务。


```
version: "3"
services:
  web:
    image: username/repo:tag
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
    resources:
      limits:
        cpus: "0.1"
        memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
networks:
  webnet:
```

这是官方例子中 `docker-compose` 的内容，一共有两个服务：一个是 `Web`，另一个是 `visualizer`。基本普通项目用到 `docker-compose` 这个阶段就够用了。假如对容器进行编排做更多的配置，就可以使用 `Kubernetes` 在不间断服务的情况下更新你的容器，并且有更强大的重启策略等，不过学习难度也比较陡峭，后面笔者会使用云提供商封装好的服务，这样可以降低学习成本。

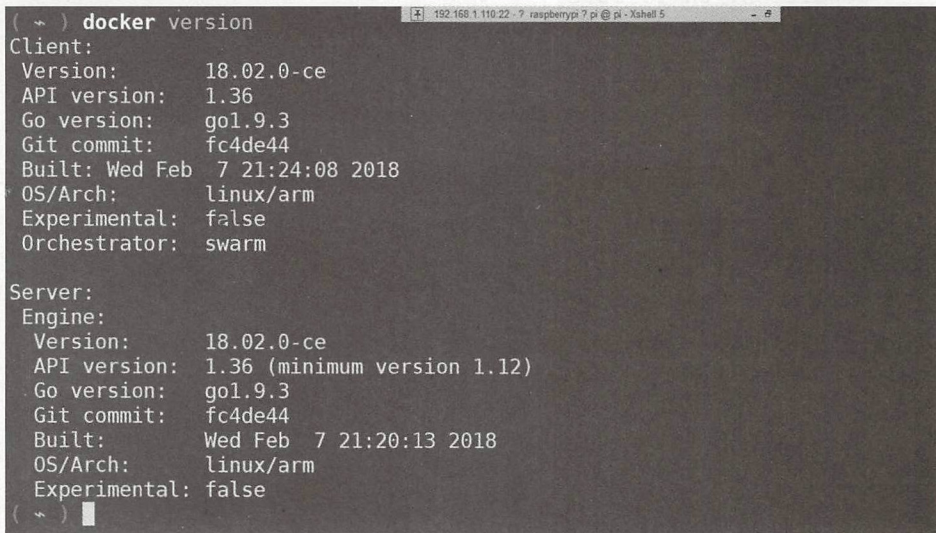
笔者不建议大家以树莓派安装 `Docker` 去尝试运行这些例子，由于树莓派架构的原因会有很多错误，这样容易打消初学者的积极性。

6.1.3 安装 Docker

我们使用 daocloud 提供的安装脚本，它会使用国内的源，这样会快一些。

```
curl -sSL https://get.daocloud.io/docker | sh
```

当安装完成之后，用 `docker version` 测试是否安装成功。



```
( ~ ) docker version
Client:
Version:      18.02.0-ce
API version:  1.36
Go version:   go1.9.3
Git commit:   fc4de44
Built: Wed Feb  7 21:24:08 2018
OS/Arch:      linux/arm
Experimental: false
Orchestrator: swarm

Server:
Engine:
Version:      18.02.0-ce
API version:  1.36 (minimum version 1.12)
Go version:   go1.9.3
Git commit:   fc4de44
Built: Wed Feb  7 21:20:13 2018
OS/Arch:      linux/arm
Experimental: false
( ~ )
```

6.1.4 使用加速器

因为构建的时候要下载基础镜像，为了加快速度所以要修改源地址。来到“加速器”按照提示，设置加速器。

或者可以通过修改 `systemd` 的配置实现加速：

```
sudo vim /lib/systemd/system/docker.service
```

```
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network-online.target docker.socket firewalld.service
Wants=network-online.target
Requires=docker.socket

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd --registry-mirror=http://b1325c57.m.daocloud.io -H fd://
ExecReload=/bin/kill -s HUP $MAINPID
LimitNOFILE=1048576
# Having non-zero Limits causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitPROC=infinity
LimitCORE=infinity
```

修改 ExecStart, 增加 -registry-mirror 参数, 在启动服务的时候指定镜像地址, 然后重载配置文件与重启服务。

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

6.1.5 下载一个基础镜像

```
docker pull Ubuntu
```

```
( ~ ) s ) system ) docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
12d2fa1549d5: Pull complete
5b66e47c4b68: Pull complete
039f9b32d330: Pull complete
cabb4c40729a: Pull complete
c83576329d6d: Pull complete
Digest: sha256:e27e9d7f7f28d67aa9e2d7540bdc2b33254b452ee8e60f388875e5b7d9b2b696
Status: Downloaded newer image for ubuntu:latest
```

6.1.6 hello world

使用以下命令。-ti 标志可以接管 docker 启动的 shell。

```
( ~ ) docker run -ti ubuntu /bin/bash
root@9ba7abfa741a:/# echo hello world
hello world
root@9ba7abfa741a:/#
```


6.2 手动构建镜像

1. Docker 架构

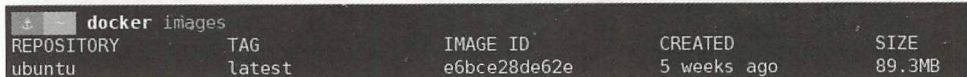
我们开发的是 B/S 架构的软件，就是浏览器作为客户端，服务器作为服务端。而 Docker 是 C/S 架构的软件，跟 MySQL 一样，它会启动一个守护进程。客户端与服务端可以运行在不同机器上，而 Docker 的客户端就是 Docker 的命令行，Docker 的服务端是一个 JSON API 服务。

对镜像每进行一次操作就会产生一个新的镜像层，即 layer。它不会显示在镜像列表中，但是它会有一个 imageID 来唯一标识它。这样的好处就是当好几个镜像有共同部分的时候，就没必要下载或者构建重复的部分。

2. 使用命令行

查看已有的镜像

通过 `docker images` 可以获得本地所有镜像，运行一个容器需要以镜像为基础，所以先学习如何管理镜像。列出的信息包括名字、标签、唯一 ID、创建时间与大小，我们把标签理解为版本号就行了。



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	e6bce28de62e	5 weeks ago	89.3MB

下载镜像

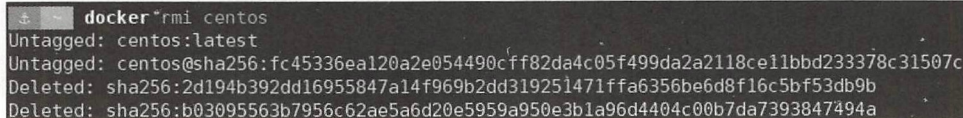
通过 `docker pull` 可以下载镜像，所有镜像都可以到 `hub.docker.com` 中找到，并且会有 README.md 告诉你如何使用该镜像。



```
docker pull centos
Using default tag: latest
latest: Pulling from library/centos
5e35d10a3eba: Downloading [=====] 25.51MB/72.98MB
```

删除镜像

使用 `docker rmi` 可以删除一个镜像，后面跟名字或者 imageId 都可以，只要能唯一标识一个镜像就行。



```
docker rmi centos
Untagged: centos:latest
Untagged: centos:sha256:fc45336ea120a2e054490cfff82da4c05f499da2a2118ce11bbd233378c31507c
Deleted: sha256:2d194b392dd16955847a14f969b2dd319251471ffa6356be6d8f16c5bf53db9b
Deleted: sha256:b03095563b7956c62ae5a6d20e5959a950e3b1a96d4404c00b7da7393847494a
```

```

$ docker rmi 2d194b392dd1
Untagged: centos:latest
Untagged: centos@sha256:fc45336ea120a2e054490cff82da4c05f499da2a2118ce11bbd233378c31507c
Deleted: sha256:2d194b392dd16955847a14f969b2dd319251471ffa6356be6d8f16c5bf53db9b
Deleted: sha256:b03095563b7956c62ae5a6d20e5959a950e3b1a96d4404c90b7da7393847494a

```

运行镜像

通过 `docker run` 可以运行一个镜像，产生一个容器，尝试安装 VIM 后退出。

```

$ docker run -ti centos /bin/bash
[root@0f69c7bb83a5 /]# yum install vim -y

```

查看所有容器

通过 `docker ps -a` 查看所有容器，不加 `a` 参数会隐藏停止的容器。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
0f69c7bb83a5	centos	"/bin/bash"	2 minutes ago	Exited (130) 28 seconds ago		vigilant_

将修改好的容器提交为镜像

通过 `docker commit` 将安装了 VIM 的 CentOS 容器制作为自己的镜像。通过 `docker images` 可以看到刚刚制作完成的镜像。

```

$ docker commit 0f69c7bb83a5 my_centos:alpha
sha256:8d6ed250b70e4eb84242a825c5c01e222c66c6a034bdb336ec61cdf5cc35bbfc

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my_centos	alpha	8d6ed250b70e	About a minute ago	171MB
centos	latest	1401c2e2000f	7 days ago	171MB
ubuntu	latest	e6bce28de62e	5 weeks ago	89.3MB

登录镜像中心

到 `hub.docker.com` 注册账户，然后通过 `docker login` 进行登录。

```

$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: yugo
Password:
Login Succeeded

```

通过 `docker push` 推送镜像，直接推送 `my_centos` 会提示没有权限。

```

$ docker push my_centos
The push refers to repository [docker.io/library/my_centos]
9a31c79fc5d3: Preparing
24bd07feb51c: Preparing
denied: requested access to the resource is denied

```


创建镜像副本

可以把 `docker tag` 当作重命名来使用，这里在名字前面加了 `yugo/`，这是 `hub.docker.com` 的命名空间。

```
➤ docker tag my_centos:alpha yugo/my_vim_centos_test
➤ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
my_centos	alpha	8d6ed250b70e	13 minutes ago	171MB
yugo/my_vim_centos_test	latest	8d6ed250b70e	13 minutes ago	171MB
centos	latest	1401c2e2000f	7 days ago	171MB
ubuntu	latest	e6bce28de62e	5 weeks ago	89.3MB

推送镜像

通过 `docker pull` 推送刚刚构建好的镜像。`docker save` 可以把镜像打包成压缩文件，`docker load` 也可以导入这样的文件，但是这样做需要自己进行网络传输。

```
➤ docker push yugo/my_vim_centos_test
The push refers to repository [docker.io/yugo/my_vim_centos_test]
9a31c79fc5d3: Pushing [=====>] 2.56kB
24bd07feb51c: Pushing [=====>] 5.359MB/171.4MB
```

如何搭建私有的镜像

笔者不建议这么做，因为非常多的云服务提供商都有免费的私有镜像服务，但是用 Docker 启动一个私有镜像服务其实也非常简单。

```
docker run -d -p 5000:5000 --restart always --name registry registry:2
docker push localhost:5000/Ubuntu
```

`d` 指定为后台启动；`p` 指定端口，即容器内的 5000 端口映射到 Docker 本机的 5000 端口；`restart` 指定重启策略，当异常退出时自动重启；`name` 可以指定容器名称。

删除一个容器

通过 `docker rm` 可以删除一个停止的容器。

```
➤ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
0f69c7bb83a5	centos	"/bin/bash"	36 minutes ago	Exited (130) 35 minutes ago		vigilant_

```
➤ docker rm 0f69c7bb83a5
```

停止运行一个容器

当我们移除一个运行的容器时会发生错误。

```
± docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
96098cc05ca5        centos              "/bin/bash"         7 seconds ago       Up 6 seconds                quizzical_banach

± docker rm 96098cc05ca5
Error response from daemon: You cannot remove a running container 96098cc05ca592863c115b9a6a7d22aa4f354445def529e0972edc12727932f7. Stop the container before attempting removal or force remove
```

通过 `docker stop` 停止之后再删除即可。

```
± docker stop 96098cc05ca5
96098cc05ca5
± docker rm 96098cc05ca5
96098cc05ca5
```

重启并重新接管 shell

首先通过 `docker run` 接管一个 `centos` shell，然后退出。当使用 `docker attach` 想要接管 shell 时它提示我们已经停止了，可以通过 `docker start` 启动之后，再使用 `docker attach` 即可。

```
± docker run -ti centos /bin/bash
[root@b695f75bbf1 /]# exit
exit
± docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b695f75bbf1        centos              "/bin/bash"         12 seconds ago      Exited (0) 7 seconds ago                eloquent_eli

± docker attach b69
You cannot attach to a stopped container, start it first

± docker start b69
b69

± docker attach b69
[root@b695f75bbf1 /]#
```

让运行的容器再开一个 shell

通过 `docker exec` 可以在容器中执行一些命令，这里通过 `exec` 再启动一个 shell 并接管它。

```
± docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b695f75bbf1        centos              "/bin/bash"         12 seconds ago      Up 7 seconds                eloquent_eli

± docker exec -ti b69 /bin/bash
[root@b695f75bbf1 /]#
```

3. 总结

通过学习上面的命令，现在已经可以使用 Docker 运行一些简单的 Web 应用了。通常来说我们是不直接手动去做的，而是通过 Dockerfile 来做这些事情，在讲解 Dockerfile 的时候，我们再对一些参数进行说明。

6.3 编写 Dockerfile 文件

编写 Dockerfile 最重要的是认识指令，所有的指令都可以在 Docker Documentation 中找到。

1. 如何通过 Dockerfile 构建镜像

```
docker build .
```

`docker build` 命令会在当前目录下找到 Dockerfile 进行构建。有的 Dockerfile 文件中指定要复制一些文件到容器中，所以需要文件上下文，命令后面会有一个点，代表当前目录。

2. FROM 指令

镜像不是凭空产生的，而是基于官方的基础镜像进行更改的，所以会有一个 FROM 指令，表示以何种镜像为基础。

```
FROM Ubuntu
```

3. RUN 命令

`run` 命令用来执行一些 shell 命令，比如安装依赖等，通常建议把命令尽量都写到一行，因为一个指令就是一个镜像层，这种安装的命令合在一个层是一种最佳实践。

```
RUN apt-get install -y vim
```

4. CMD 与 ENTRYPOINT 命令

假如某个镜像有 CMD 命令，且这个镜像叫 `my_node`：

```
CMD node index.js
docker run my_node
```

当我们通过 `docker run` 直接启动容器的时候，它会执行 `node index.js` 命令：

```
docker run -ti my_node /bin/bash
```

假如在启动容器的时候加上了 `/bin/bash` 参数，则参数会替换 `node index.js` 命令。

ENTRYPOINT 则不会被替换，它会让镜像如同一个可执行程序一样。例如：

```
ENTRYPOINT ["/bin/echo"]
```



```
docker run my_node hello world
```

它会输出 `hello world`，而不是替换掉命令。

5. VOLUME 指令

```
VOLUME ["/code"]  
WORKDIR /code  
RUN node index.js  
docker run -v $PWD/app:/code
```

它会把当前目录 `app` 下的所有文件同步到镜像中，默认 Docker 容器中的数据都是只读的，要想让它可写，并且可以共享给其他容器就必须创建 `volume`。

6. COPY 与 ADD

这两个命令是把文件复制到容器中，两者类似，相比较而言，`ADD` 读取远程网络文件更加优秀，一般使用 `COPY` 就够用了。

```
COPY . /code
```

7. EXPORT

将本机的 80 端口映射到容器内部的 8080 端口：

```
EXPORT 80 8080
```

8. WORKDIR

用于切换目录，类似于 `cd` 命令：

```
WORKDIR /code  
RUN npm run start
```

9. 运行一个 Server 实例

- 创建 `Dockerfile`

对于笔者来说，小写更加容易阅读。

```
from node:slim
```



```
copy . /code

workdir /code

expose 8080

run npm install

cmd node index.js
```

上面的内容表示从 node 镜像的 slim 版本构建，slim 版本是运行 node 应用最精简的容器。将当前目录复制到容器的 /code 目录下，进入/code 目录下，暴露出容器的 8080 端口，安装依赖，运行 index.js 文件。

- 创建 index.js

```
const server = require('server')
const { get } = server.router

server({ port: 8080 }, [
  get('/', ctx => 'Hello Docker')
])
```

- 安装依赖

```
npm init -y
npm i server -D
```

- 忽略 node_modules

创建 .dockerignore，它会忽略其中所列出来的文件。

```
node_modules
```

- 进行构建

通过 t 参数可以指定名字。

```
docker build . -t my_server
```

```
± /c/dd docker build . -t my_server
Sending build context to Docker daemon 40.76MB
Step 1/4 : from node
latest: Pulling from library/node
4176fe04cefe: Downloading [==>] 3.152MB/52.61MB
851356ecf618: Downloading [=====>] 10.87MB/19.27MB
6115379c7b49: Downloading [=====>] 10.47MB/43.25MB
aaf7d781d601: Waiting
936f8420f2e4: Waiting
eab82fe5fcf4: Waiting
77bc0fa5883e: Waiting
908773ec9e8f: Waiting
```

- 运行

```
docker run -p 8080:8080 my_server
```

- 测试

```
curl localhost:8080
```

```
± docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                    NAMES
e2aa5a78de51   my_server "/bin/sh -c 'node in..." 5 seconds ago Up 4 seconds   0.0.0.0:8080->8080/tcp   quirky_gg
ldstine
± curl localhost:8080
Hello Docker!
```

6.4 Docker Compose

docker-compose 使用的配置文件是 yaml 格式，该格式基于缩进的配置文件语法，想了解 yaml 可以到 <https://nodelover.me/course/config-file> 观看笔者录制的视频。

6.4.1 安装 docker-compose

1. pip 安装

```
sudo pip3 install docker-compose
```

2. 手动下载

```
curl -L https://github.com/docker/compose/releases/download/1.19.0/
docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

由于版本一直在更新，所以可以到 <https://github.com/docker/compose/releases> 的发布页面查看当前发布的版本。

6.4.2 命令行接口

1. build

`docker-compose.yml` 中描述了多个容器的属性。假如容器所需要的镜像需要自定义，那么要有一个 `build` 操作，将自定义的 `Dockerfile` 先“`build`”成镜像再运行。假如需要的镜像不存在，则会通过 `pull` 从镜像仓库去拉取。

```
docker-compose build
```

2. up

`up` 可以启动 `docker-compose.yml` 中所有的服务，而 `-d` 可以让服务在后台运行，也就是说，日志信息不会输出到控制台。

```
docker-compose up -d
```

3. logs

如果想要查看后台启动的服务的日志，则可以使用 `logs` 命令。

```
docker-compose logs
```

4. down

把所有通过 `up` 命令创建的容器停止并销毁。

```
docker-compose down
```

5. ps

查看通过 `docker-compose` 启动的容器的详细信息。

```
docker-compose ps
```

6. stop

停止通过 `docker-compose up` 启动容器。

```
docker-compose stop
```

7. start

启动被 stop 命令停止的容器：

```
docker-compose start
```

8. restart

重启容器：

```
docker-compose restart
```

9. exec

在某个容器中执行命令，默认会接管一个 shell：

```
docker-compose exec web sh
```

6.4.3 Egg.js 简单实例

1. 初始化项目

```
egg-init --type=simple docker-sample
```

2. 安装依赖

```
cd docker-sample  
npm install
```

3. 创建 Dockerfile

先将 package.json 复制进去进行安装，之后将代码文件复制到其中。

```
FROM node:slim  
RUN mkdir /myapp  
WORKDIR /myapp  
COPY package.json /myapp/package.json  
RUN npm install --registry=https://registry.npm.taobao.org  
COPY . /myapp
```


4. 创建 docker-compose.yml

顶级的配置项有如下几项：

- version, 描述当前的版本；
- services, 描述提供服务的容器；
- networks, 描述服务之间的网络连接；
- volumes, 描述持久化存储卷；
- configs, 描述配置文件所在的地方；
- secrets, 描述密码文件所在的地方；
- x-varname, 以 x-开头的一些扩展字段。

不要觉得 docker-compose 很难,从某种意义上说,它其实就像 npm 的一个 package.json 一样。当你使用得多了,就会知道它的配置项了,它所有的配置项可以在 <https://docs.docker.com/compose/compose-file> 上找到,这里我们只把用到的配置项阐述一下。

```
version: '3'
services:
  web:
    build: .
    command: npm run dev
    volumes:
      - ../myapp
    ports:
      - "7001:7001"
```

version 描述了当前描述语法的版本,而 services 指明了提供服务的容器。web 是一个容器,build 配置项告诉我们,Web 镜像从当前目录的 Dockerfile 进行构建,且将本地当前路径与容器的 myapp 目录保持同步,本机的 7001 端口映射到容器内部的 7001 端口。

假如是生产环境,则直接去掉 volumes,不同步就可以了,因为在配置镜像的时候已经复制过一次了。

上面的 yml 等同于下面的 JSON 对象,以横杆开头的都是数组中的元素。

```
{
  version: '3',
  services: {
    web: {
```





```
    build: '.',
    command: 'npm run dev',
    volumes: ['./myapp'],
    ports: ['7001:7001']
  }
}
```

5. 构建镜像

```
docker-compose build
```

6. 启动

```
docker-compose up
```

```
~/w/docker-sample $ docker-compose up
Recreating dockersample_web_1 ... done
Attaching to dockersample_web_1
web_1 |
web_1 | > docker-sample@1.0.0 dev /myapp
web_1 | > egg-bin dev
web_1 |
web_1 | 2018-03-09 06:47:23,263 INFO 26 [master] node version v9.7.1
web_1 | 2018-03-09 06:47:23,273 INFO 26 [master] egg version 2.4.1
web_1 | 2018-03-09 06:47:25,690 INFO 26 [master] agent_worker#1:32 started (2397ms)
web_1 | 2018-03-09 06:47:29,792 INFO 26 [master] egg started on http://127.0.0.1:7001 (6518ms)
```

7. 测试

7001 端口会返回正常的结果。

```
curl localhost:7001
```

```
~ $ curl localhost:7001
hi, egg
```

6.4.4 增加服务

后端有数据库是非常正常的一件事情，现在我们来添加一个 MySQL 数据库。

1. 安装依赖

```
npm install egg-sequelize mysql2 -S
```





2. 开启依赖

配置 plugin.js 与 config.default.js:

```
exports.sequelize = {
  enable: true,
  package: 'egg-sequelize'
}
config.sequelize = {
  host: 'db',
  dialect: 'mysql',
  database: 'app',
  username: 'root',
  password: '888888'
}
```

db 就是 docker-compose 服务的名称，Docker 容器中会知道如何找到它。

3. 修改 docker-compose.yml

在镜像仓库中找到你要添加的镜像，查看它的文档，通常都会告诉你如何使用，比如 MySQL。

```
version: '3.2'
services:
  web:
    build: .
    command: npm run dev
    restart: always
    volumes:
      - ../myapp
    ports:
      - "7001:7001"

  db:
    image: mysql:5.6
    restart: always
    volumes:
      - "db-data:/var/lib/mysql"
```





```
environment:
  MYSQL_ROOT_PASSWORD: 8888888
  MYSQL_DATABASE: app

adminer:
  image: adminer
  restart: always
  ports:
    - 9000:8080

volumes:
  db-data:
```

`image` 用于指定基础镜像，冒号后面跟的是版本号。添加 `volumes` 是为了让数据持久化，也就是下一次通过 `up` 启动时不会丢失数据。`restart` 是为了让启动失败后会一直重启，因为 Web 连不上数据库会异常退出，虽然可以使用 `depends_on` 选项来指定依赖，但是 `depends_on` 只能确保容器的启动顺序，并不能确保 Web 启动的时候，MySQL 容器的服务做好了接受连接的准备。

`MYSQL_DATABASE` 与 `MYSQL_ROOT_PASSWORD` 是环境变量，MySQL 镜像的启动脚本会根据设置的环境变量来设置 `root` 的密码和创建数据库名称。

4. 启动

```
docker-compose up
```

可以成功地访问 7001 和 9000 端口的应用如图 6-1、图 6-2 所示。

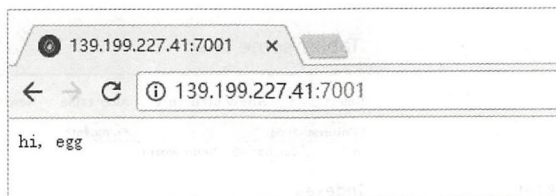


图 6-1



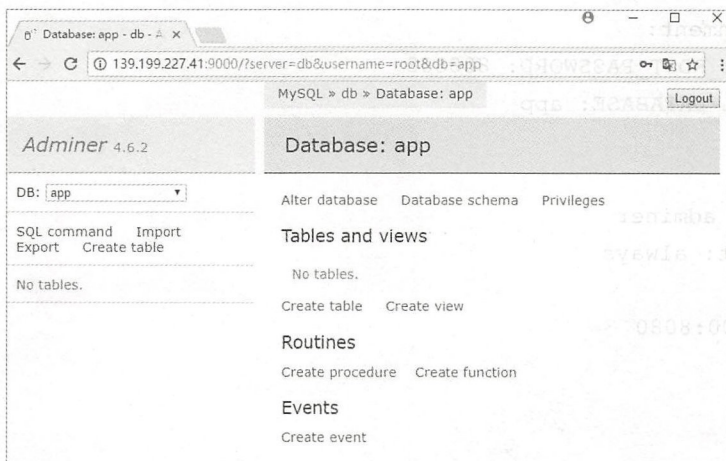


图 6-2

5. 确认数据是否持久化

```
^CGracefully stopping... (press Ctrl+C again to force)
Stopping dockersample_web_1 ... done
Stopping dockersample_adminer_1 ... done
Stopping dockersample_db_1 ... done
[root@VM_138_125_centos docker-sample]# docker-compose start
Starting web ... done
Starting adminer ... done
Starting db ... done
```

首先按 Ctrl + C 组合键退出，然后通过 `docker-compose start` 启动，发现现在以后台的方式运行这些应用了，假如还想要查看日志，则可以通过 `docker-compose logs -f` 来追踪日志，让日志一直在终端输出。

进入 Adminer 随便创建一个表，笔者这里创建了一个叫作 `some one` 的表，如图 6-3 所示。

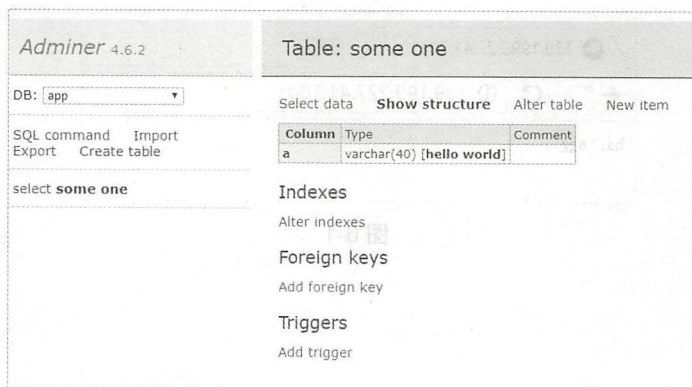


图 6-3





先进行 stop 操作再进行 start 操作。

```
[root@VM_138_125_centos docker-sample]# docker-compose stop
Stopping dockersample_web_1 ... done
Stopping dockersample_adminer_1 ... done
Stopping dockersample_db_1 ... done
[root@VM_138_125_centos docker-sample]# docker-compose start
Starting web ... done
Starting adminer ... done
Starting db ... done
```

先进行 down 操作再进行 up 操作。

```
[root@VM_138_125_centos docker-sample]# docker-compose down
Stopping dockersample_web_1 ... done
Stopping dockersample_adminer_1 ... done
Stopping dockersample_db_1 ... done
Removing dockersample_web_1 ... done
Removing dockersample_adminer_1 ... done
Removing dockersample_db_1 ... done
Removing network dockersample_default
[root@VM_138_125_centos docker-sample]# docker-compose up
```

数据都会依旧存在。

6. 数据库迁移

直接使用 Adminer 的导出功能即可，非常方便，如图 6-4 所示。

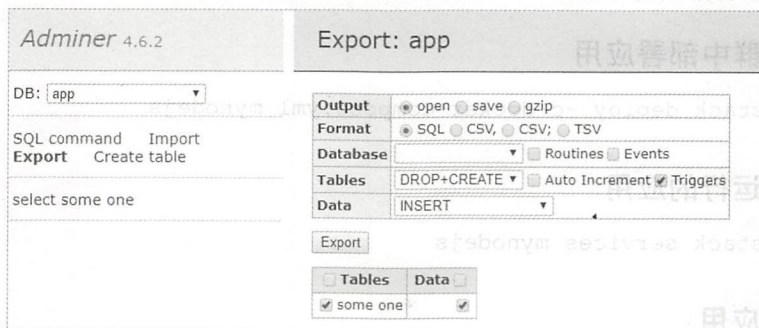


图 6-4

7. 清空

加上 volumes 会清空存储卷：

```
docker-compose down --volumes
```





6.5 集群

6.5.1 Docker 集群

之前启动的应用都是在一台机器上，一台机器的性能再好也是有局限的，就像一根筷子与十根筷子的区别。Docker 之间可以进行连接，就像网络的节点一样，可以形成联动。构建 Docker 集群的前提就是，Docker 所在的机器网络可以相互连接。

6.5.2 集群初始化

1. 初始化主节点

```
docker swarm init --advertise-addr xxx.xxx.xxx.xxx
```

xxx.xxx.xxx.xxx 代表的是本机的 IP 地址，它会给出一个 join 的命令，附带一个 Token。

2. 连接主节点

```
docker swarm join --token xxxxx xxx.xxx.xxx.xxx:2377
```

根据上一步骤给出的 Token，与主节点 IP 地址进行连接。

3. 在集群中部署应用

```
docker stack deploy -c docker-compose.yml mynodejs
```

4. 查看运行的应用

```
docker stack services mynodejs
```

5. 删除应用

```
docker stack rm mynodejs
```

6. docker-compose.yml

对于集群模式，会支持服务的 deploy 配置项。所有的 deploy 配置项都可以在 Compose file version 3 reference | Docker Documentation 上找到。





```
version: '3'
services:
  redis:
    image: redis:alpine
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
```

上面代码的意思是，通过 `replicas` 指定启动 6 个容器。`update_config` 指定更新策略，`parallelism` 指定每组更新 2 个，完成之后延迟 10 秒再更新下一组。`restart_policy` 指定更新的策略，而 `condition` 则是条件，表示失败重启。

6.5.3 实例

1. 添加代码

新建 `index.js`:

```
const os = require('os')

const http = require('http')

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end(os.hostname())
})

server.listen(3002, () => {
  console.log('start')
})
```





2. 添加配置

创建 docker-compose.yml:

```
version: '3'
services:
  web:
    image: mynode
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "3002:3002"
    networks:
      - webnet
networks:
  webnet:
```

集群模式不支持构建镜像，所以只能自己构建。resources 的 limits 是对系统资源做一些限定，每个容器能使用 10% 的 CPU 与 50MB 的内存。

3. 初始化节点

docker swarm init

```
[root@VM 138 125 centos docker-sample]# docker swarm init
Swarm initialized: current node (kqyc3vboalslh4d8fp4jhif9q) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4b18psttp7pm3f12amgsrf219t3otdgeoipdk6erko8qx5gp9o-7e94c45ypgqscpcpme32j9j25 10.104.138.125:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

因为笔者只有一台机器，所以就用一台机器来测试，一台机器也是可以开启 swarm 模式的。





4. 构建镜像

添加 Dockerfile:

```
FROM node:slim
RUN mkdir /code
COPY index.js /code/index.js
WORKDIR /code
EXPOSE 3002
CMD node index.js

docker build . -t mynode
```

```
[root@VM_138_125_centos workspace]# docker build . -t mynode
Sending build context to Docker daemon 98.58MB
Step 1/6 : FROM node:slim
--> cebc261de278
Step 2/6 : RUN mkdir /code
--> Using cache
--> 37bc74335bae
Step 3/6 : COPY index.js /code/index.js
--> Using cache
--> 44e54ee0a99d
Step 4/6 : WORKDIR /code
--> Using cache
--> 4fc42baf2df3
Step 5/6 : EXPOSE 3002
--> Using cache
--> c1d509fa3f2b
Step 6/6 : CMD node index.js
--> Using cache
--> 0fe74e6423fb
Successfully built 0fe74e6423fb
Successfully tagged mynode:latest
```

5. 运行

```
docker stack deploy -c docker-compose.yml node
```

node 是这个集群的名字。

```
[root@VM_138_125_centos workspace]# docker stack deploy -c docker-compose.yml node
Creating network node_webnet
Creating service node_web
```

6. 查看服务

```
docker stack services node
```



```
[root@VM_138_125_centos workspace]# docker stack services node
```

ID	NAME	MODE	REPLICAS	IMAGE	PORTS
ijfu0wgk7mv8	node_web	replicated	5/5	mynode:latest	*:3002->3002/tcp

7. 访问 Web

结果如图 6-5 所示。



图 6-5

笔者测试的时候，是隔了几秒钟之后再刷新的，得到的是不同的主机名，说明在不同的机器上，可以得到不同的主机名。但是持续快速刷新主机名则会保持不变，而使用笔者之前测试的 yugo/py 镜像则不会存在这样的问题。假如读者想要测试，则通过 pull 拉取该镜像，然后将导出的端口改为 80 即可。这种差异行为可能是由 TCP 连接的释放策略造成的。

6.6 持续部署

本节我们利用 Git 的钩子来实现部署项目。

6.6.1 部署主机免密码登录

首先准备一台部署的主机，笔者这里使用之前的树莓派作为部署的主机，IP 地址是本地的 192.168.1.149，通过 ssh-copy-id 把本机的公钥复制到部署主机上，这样我们就可以免密码登录部署主机。

```
ssh-copy-id pi@192.168.1.149
```

当我们看到以下提示内容，代表部署已经完成了，可以通过 ssh 测试一下。

```
➔ 8-6 git:(master) X ssh-copy-id pi@192.168.1.149
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/Users/yugo/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install all the new keys
pi@192.168.1.149's password: .

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'pi@192.168.1.149'"
and check to make sure that only the key(s) you wanted were added.
```


6.6.2 客户端钩子

1. 搭建项目环境

```
npm init -y
```

安装依赖，pm2 用来部署项目，husky 是添加 git 钩子的工具。

```
npm install pm2 express husky@next -S
```

修改以下 package.json，添加钩子 post-commit，表示“commit”信息之后，就会执行 deploy.sh 脚本，我们在这个脚本中部署项目。

git 其实还提供了很多其他钩子，主要分为客户端钩子和服务端钩子，这里我们直接使用客户端钩子。

```
"scripts": {
  "start": "pm2 restart index.js"
},
"husky": {
  "hooks": {
    "post-commit": "deploy.sh"
  }
},
```

添加 deploy.sh，该脚本中做了三件事，压缩项目文件、上传项目文件、在部署机器中解压并重启项目。

```
#!/bin/bash

tar -zcvf ../node_project.tar.gz --exclude ./node_module .

scp -r ../node_project.tar.gz pi@192.168.1.149:~/workspace

ssh pi@192.168.1.149 "cd ~/workspace ; tar -zxvf node_project.tar.gz; npm
run start"
```

给 deploy.sh 添加执行权限：

```
chmod a+x ./deploy.sh
```


2. 初始化仓库

```
git init
git add .
git commit -m 'init'
```

```
Use --update-env to update environment variables
[PM2][ERROR] Process index.js not found
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	user	watching
Use `pm2 show <idname>` to get more details about an app										
npm ERR! code ELIFECYCLE										
npm ERR! errno 1										
npm ERR! 8-6@1.0.0 start: `pm2 restart index.js`										
npm ERR! Exit status 1										
npm ERR!										
npm ERR! Failed at the 8-6@1.0.0 start script.										
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.										
npm ERR! A complete log of this run can be found in:										
npm ERR! /home/pi/.npm/_logs/2018-04-24T06_34_37_182Z-debug.log										
husky > post-commit hook failed (add --no-verify to bypass)										
[master a56aeb] init										
1 file changed, 1 insertion(+), 1 deletion(-)										

虽然执行了脚本，但是似乎并没有部署成功，因为第一次运行，服务器上没有任何任务，所以 restart 会失败，我们首先执行 ssh 到部署机器，创建服务。

```
~/workspace on master x ls
deploy.sh* index.js node_modules/ package-lock.json
docker-sample/ mysql/ node_project.tar.gz package.json
~/workspace on master x npx pm2 start index.js
[PM2] Starting /home/pi/workspace/index.js in fork_mode (1 instance)
[PM2] Done.
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	user	watching
index	0	fork	14255	online	0	0s	59%	14.2 MB	pi	disabled

Use `pm2 show <idname>` to get more details about an app

进入~/workspace 执行以下命令：

```
npx pm2 start index.js
```

然后回到本机，执行以下 deploy.sh 脚本：

```
./deploy.sh
```

```
> 8-6@1.0.0 start /home/pi/workspace  
> pm2 restart index.js
```

```
Use --update-env to update environment variables.  
[PM2] Applying action restartProcessId on app [index.js](ids: 0)  
[PM2] [index](0) ✓
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	user	watching
index	0	fork	14359	online	1	0s	83%	15.5 MB	pi	disabled

```
Use `pm2 show <id|name>` to get more details about an app
```

现在我们可以看到部署成功了。修改一下代码后再次 commit，会发现依然可以部署成功。

6.6.3 使用服务端钩子进行部署

这一次我们使用 post-receive 钩子，当服务端接收 push 的时候执行，husky 服务端的钩子笔者测试没有成功，所以我们使用 git 原生的钩子机制。

1. 初始化服务器仓库

```
mkdir node_project && cd node_project  
git init -bare
```

在 node_project/hooks 下新建 post_receive 文件，当我们推送成功之后就会执行这个文件的内容，这个执行文件的环境可以随意修改成 #!/usr/bin/env node、#!/usr/bin/env python、#!/usr/bin/env perl，这样你就可以写 Node 脚本、Python 脚本、Perl 脚本等，这里我们简单地写 bash 脚本就好，内容如下：

```
#!/bin/bash
```

```
rm -rf /home/pi/work
```

```
git clone /home/pi/node_project /home/pi/work
```

```
cd /home/pi/work
```

```
npm install
```

```
npx pm2 delete all
```

```
npx pm2 start index.js
```

添加执行权限：

```
chown a+x ./post-receive
```

2. 初始化本地仓库

```
mkdir node_project && cd node_project
git init
git add .
git commit -m 'init'
```

然后将之前的代码复制过来，把 husky 的钩子删除即可。

```
git remote add origin pi@192.168.1.149:/home/pi/node_project
git push -u origin master
```

6.6.4 使用 shipit

安装依赖

```
npm i shipit-cli shipit-deploy -D
```

随便创建一个项目，使用 shipit 的 deploy 模块部署，其实是使用 git 进行部署的，因为前面我们在服务端创建了 git 仓库，现在我们使用上一节的 git 仓库进行部署。

当然，shipit 提供了 copyToRemote 方法，这样其实也可以将文件从本地复制到部署服务器上。shipit 使用 git 部署的时候会保存多个版本，而在 deployTo 配置文件下的 current 文件夹中则是当前的版本。

在项目下创建 shipitfile.js，shipit.remote 方法默认是在用户目录下，所以我们要通过 cwd 配置项来修改执行命令的目录。

```
module.exports = shipit => {
  require('shipit-deploy')(shipit)
  shipit.initConfig({
    default: {
      deployTo: '/home/pi/work2',
      repositoryUrl: 'pi@192.168.1.149:/home/pi/node_project',
    },
  },
```



```

    staging: {
      servers: 'pi@192.168.1.149',
    },
  })
  shipit.task('deploy:finish', async () => {
    const config = {
      cwd: '/home/pi/work2/current'
    }
    const run = async (shell) => shipit.remote(shell, config)
    await run('pwd')
    await run('npm install')
    await run('npx pm2 delete all')
    await run('npx pm2 start index.js')
  })
}

```

运行命令:

```
npx shipit staging deploy
```

```

@192.168.1.149 [PM2] Applying action deleteProcessId on app [all](ids: 0)
@192.168.1.149 [PM2] [index](0) ✓
@192.168.1.149
@192.168.1.149 | App name | id | mode | pid | status | restart | uptime | cpu | mem | user | watchin
g |
@192.168.1.149 |
@192.168.1.149 Use `pm2 show <id|name>` to get more details about an app
Running "npx pm2 start index.js" on host "192.168.1.149".
@192.168.1.149 [PM2] Starting /home/pi/work2/releases/20180424091759/index.js in fork_mode (1 instan
ce)
@192.168.1.149 [PM2] Done.
@192.168.1.149
@192.168.1.149 | App name | id | mode | pid | status | restart | uptime | cpu | mem | user |
watching |
@192.168.1.149 |
@192.168.1.149 | index | 0 | fork | 7039 | online | 0 | 0s | 80% | 14.1 MB | pi |
disabled |
@192.168.1.149 |
@192.168.1.149 Use `pm2 show <id|name>` to get more details about an app
Finished 'deploy:finish' after 1.08 min
Finished 'deploy' [ deploy:init, deploy:fetch, deploy:update, deploy:publish, 'deploy:clean, deploy:f
inish ]

```


当看到 pm2 的输出的时候，代表我们已经部署成功了。当成功提交代码之后，在部署时运行该命令即可，并且通过 `npx shipit staging rollback` 命令还可以回退版本。

6.6.5 使用 Ansible 部署

Ansible 是专业的运维工具，我们部署这样一个简单的项目其实也不难。

1. 安装

笔者安装的时候并没有自带 pip，所以我们安装完 Python，之后还要手动安装 pip。

```
brew install python
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python ./get-pip.py
```

假如在安装 Python 的时候无法 link，是因为没有文件与权限的原因，可以使用下面命令来解决问题。

```
sudo mkdir /usr/local/Frameworks
sudo chown $(whoami):admin /usr/local/Frameworks
brew link python
pip install ansible
```

2. 搭建环境

如图 6-6 所示建立目录结构，node.retry 是自动生成的，不用创建。

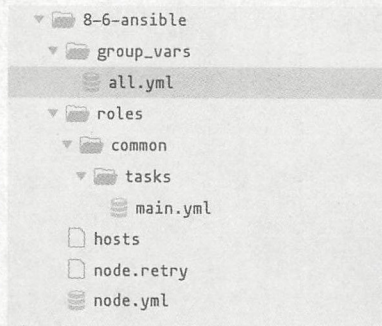


图 6-6

在 hosts 文件中定义我们要执行任务的 IP 地址，这些服务器都应该通过 ssh-copy-id 进行配置，并且实行免密码登录。

```
[server]
192.168.1.149
```

创建 `node.yml`，`hosts` 表示要连接的服务器，`remote_user` 表示要登录的用户，`roles` 表示要进行操作的角色，而 `common` 表示 `roles` 文件夹中 `common` 文件夹里的所有内容。

```
---
- name: deploy my node application
  hosts: server
  remote_user: pi
  roles:
    - common
```

创建 `main.yml`，在这个文件中，我们使用 `ansible` 写好的一些模块来安装东西、下载东西、重启服务等。在这个文件中，我们首先删除上一个版本的代码，然后通过 `git` 下载源码，通过 `npm` 安装依赖，最后通过 `pm2` 启动服务。

所有的模块都可以在文档中找到，并且都会有使用的例子，把代码复制过来改一下就可以了。

文件的结构可以参考 <https://github.com/ansible/ansible-examples> 的这个项目，笔者就是参考了这个项目中的 `lamp_simple` 例子。

```
---
- name: remove prev version
  # shell: 'rm -rf /home/pi/work3'
  file:
    path: /home/pi/work3
    state: absent

- name: fetch git source code
  git:
    repo: "{{ repo_url }}"
    dest: /home/pi/work3

- name: install dependencies
  npm:
    executable: /usr/bin/npm
    path: /home/pi/work3
```


4. pm2 错误

```

/home/pi/.pm2/logs/index-error-0.log last 15 lines:
01index | at Function.Module._resolveFilename (module.js:555:15)
01index | at Function.Module._load (module.js:482:25)
01index | at Function.Module.runMain (module.js:701:10)
01index | at startup (bootstrap_node.js:194:16)
01index | at bootstrap_node.js:618:3
01index | module.js:557
01index | throw err;
01index | ^
01index | Error: Cannot find module '/home/pi/workspace/node_modules/pm2/lib/ProcessContainerFork.js'
01index | at Function.Module._resolveFilename (module.js:555:15)
01index | at Function.Module._load (module.js:482:25)
01index | at Function.Module.runMain (module.js:701:10)
01index | at startup (bootstrap_node.js:194:16)
01index | at bootstrap_node.js:618:3

```

笔者把之前几节部署的内容删除了，当出现这样一个错误的时候，说明缓存有问题，登录部署服务器，删除之前的配置与缓存。

```
rm -rf ~/.pm2
```

6.7 持续集成

这里我们以 GitLab 为例，假如想要尝试 gogs 与 fowci，则可以观看笔者录制的视频：<https://nodelover.me/status/video/11>。本节还是不可以全部自动化，有一部分还需要手动操作，因为是第一次运行，跟之后的重启还有一些不同，所以第一次的时候还是要手动做一些事情。当然大家也可以使用其他的持续工具，比如 DaoCloud，但都比较贵。

1. 创建 Docker 镜像服务

进入腾讯云的容器服务中的镜像仓库中，创建一个命名空间。

单击“重置密码”按钮，稍后我们上传镜像时需要认证，然后单击“新建”按钮，如图 6-8 所示。

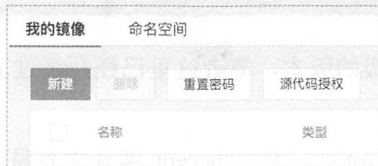


图 6-8

新建完成之后可以看到如图 6-9 所示的界面。



图 6-9

2. 创建仓库

注册 gitlab.com 的账号，并新建一个项目。

将之前 miao 的代码进行复制过来，然后做一些修改，大家没必要照着笔者所说的去做，因为有可能一些小细节会漏掉，导致你需要进行各种调试。笔者调试了一天，也提交了好几个 issues，才解决了这些问题，直接到 <https://gitlab.com/MiYogurt/miao> 上复制，通过以下命令切换到第一个版本即可。

```
git checkout v1
```

3. 修改

现在笔者来讲解一下修改了哪些代码。首先升级了版本，因为距开始初始化项目的时候已经过去了几个月，有一些版本需要升级。

```
npm install -g npm-check
```

```
npm-check -u
```

```
→ miao npm-check -u
? Choose which packages to update. (Press <space> to select)

Missing. You probably want these.
> ○ bcryptjs missing      2.4.3 https://github.com/dcodeIO/bcrypt.js#readme
Space to select. Enter to start upgrading. Control-C to cancel.
```

这样你就可以选取想要升级的版本，笔者这里已经升级过了，所以需要升级的版本就很少了。

然后将 `bcrypt` 修改为 `bcryptjs`，因为 `bcrypt` 编译太容易失败了，所以我们用纯 JS 实现的版本。

添加启动脚本，因为第一次运行的时候要创建数据库，所以要运行 `migration`，`NODE_ENV` 的值必须是 `production`，不能是 `prod` 或者其他，因为配置文件里的名字就是 `production`，否则

你会得到一个版本必须是 v4.0.0 的错误，让人莫名其妙。之后通过 egg-scripts 在前台启动应用即可。

```
"migrate": "NODE_ENV=production sequelize db:migrate",
"docker": "npm run migrate && NODE_ENV=prod egg-scripts start --title=egg-server-miao",
```

假如你多次创建多个版本的 MySQL 数据库，则 volume 会发生冲突，使用下面的命令，清空即可。

```
docker volume prune
```

添加了构建文件 Dockerfile 与 docker-compose.yml。

在 Dockerfile 中，先复制 package.json 的好处就是在 package.json 没有更改的时候就可以使用缓存，没必要每次都进行构建。

```
FROM node:9.2.0

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY package.json /usr/src/app/

# RUN npm i --production

# RUN npm i --registry=https://registry.npm.taobao.org

RUN yarn

COPY . /usr/src/app

EXPOSE 7001

CMD npm run docker
```

因为 docker-compose.yml 中设置了环境变量，所以 config 中的文件也要做相应的变更，具体请到 GitLab 上查看源码，假如不希望把端口暴露出来，则可以把除 Web 之外的服务的

ports 都删除。

```
version: '3.1'
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
    restart: always
    depends_on:
      - cache
      - db
    networks:
      - docker_miao
    ports:
      - 7001:7001

  cache:
    image: redis:3.2-alpine
    command: redis-server --appendonly yes --requirepass password
    volumes:
      - egg-redis:/data
    networks:
      - docker_miao
    ports:
      - 6379:6379

  db:
    image: mysql:5.7
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=miao
    volumes:
      - egg-mysql:/var/lib/mysql
    networks:
      - docker_miao
    ports:
```

```
- 3306:3306
```

```
adminer:
```

```
image: adminer
```

```
restart: always
```

```
networks:
```

```
- docker_miao
```

```
ports:
```

```
- 8080:8080
```

```
volumes:
```

```
egg-mysql:
```

```
egg-redis:
```

```
networks:
```

```
docker_miao:
```

4. 本地测试

```
docker-compose build
```

```
docker-compose up
```

进入 admin 页面, 这个时候你可能看到的是空白页, 因为表有几个名字有错误, 要改成大写才行, 这个错误在第二次提交时会修复, 这里大家可以打开 localhost:8080, 进入 Adminer, 输入用户名 root 和 密码 password, 进入“管理”页面, 把小写表名字的首字母通过 Alter Table 都修改成大写的, 如图 6-10 所示。

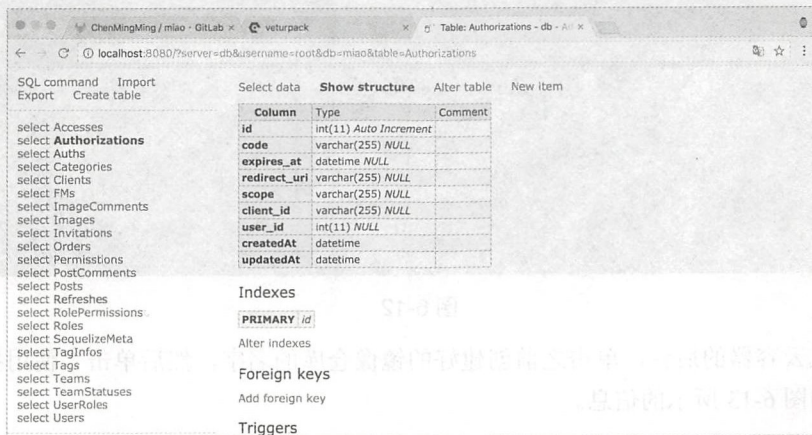


图 6-10

这样就可以看到如图 6-11 所示的页面。

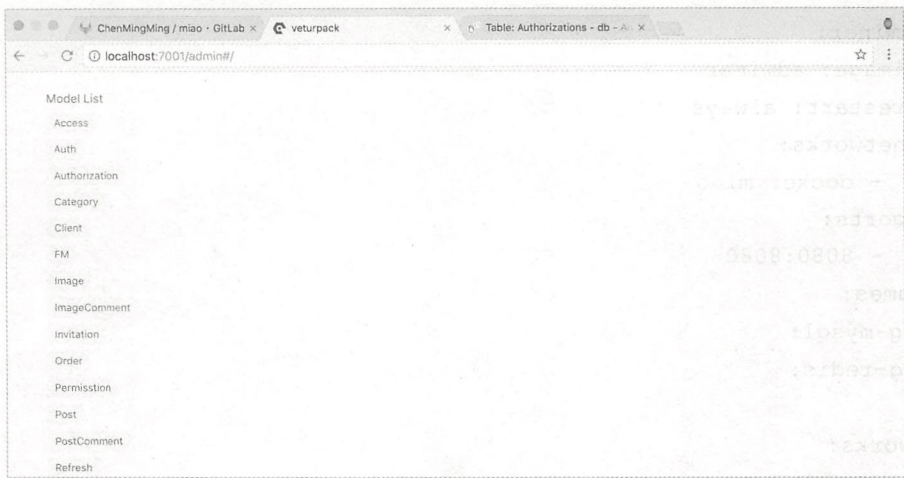


图 6-11

完成之后，清除容器相关的数据：

```
docker-compose down
```

然后将镜像上传到腾讯云容器。此时镜像中应该会有构建好的 `miao_web` 镜像，如图 6-12 所示。

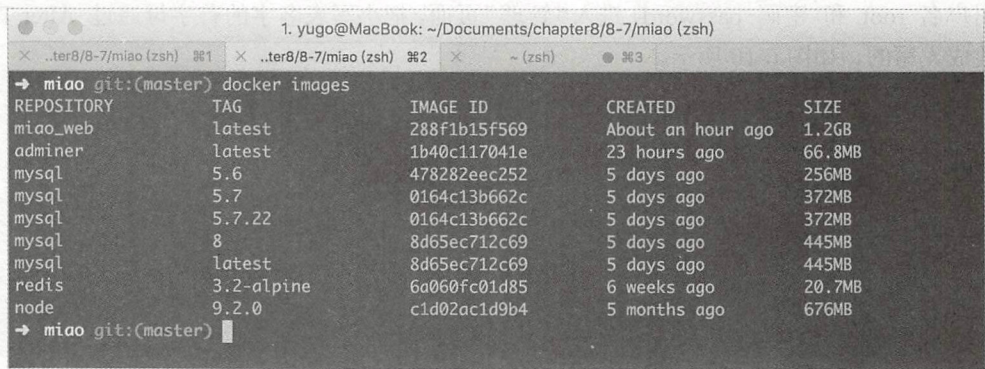


图 6-12

进入腾讯云容器的后台，单击之前创建好的镜像仓库的名字，然后单击“使用指引”选项就可以看到如图 6-13 所示的信息。



图 6-13

按照它给出的提示，我们将镜像推送到腾讯云容器的镜像仓库中，如图 6-14 所示。

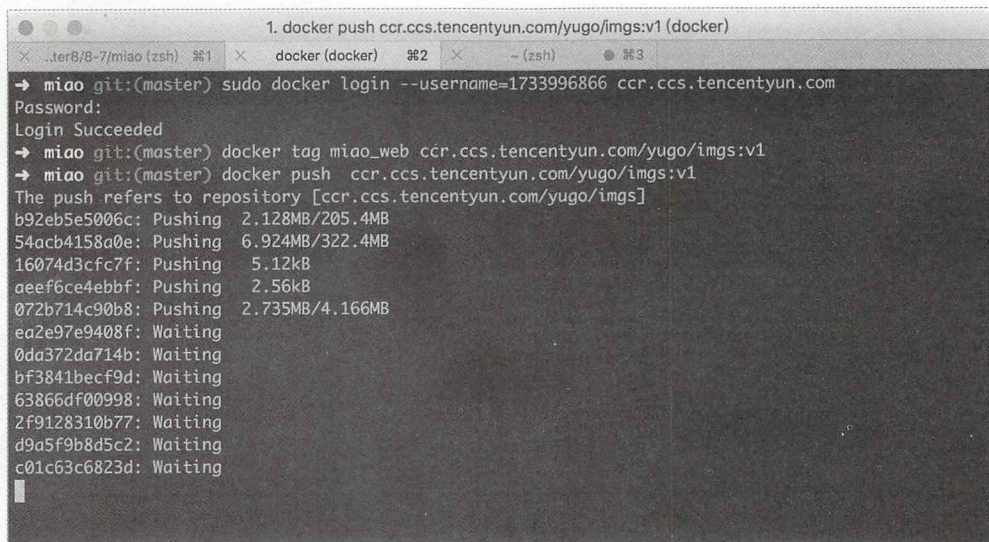


图 6-14

5. 让腾讯云帮我们构建

其实 GitLab 为每一个项目都提供了容器镜像仓库，我们先使用腾讯云，回到镜像仓库页

面，单击“源码授权”按钮，如图 6-15 所示。



图 6-15

在 GitLab 中生成个人的访问令牌，如图 6-16 所示。

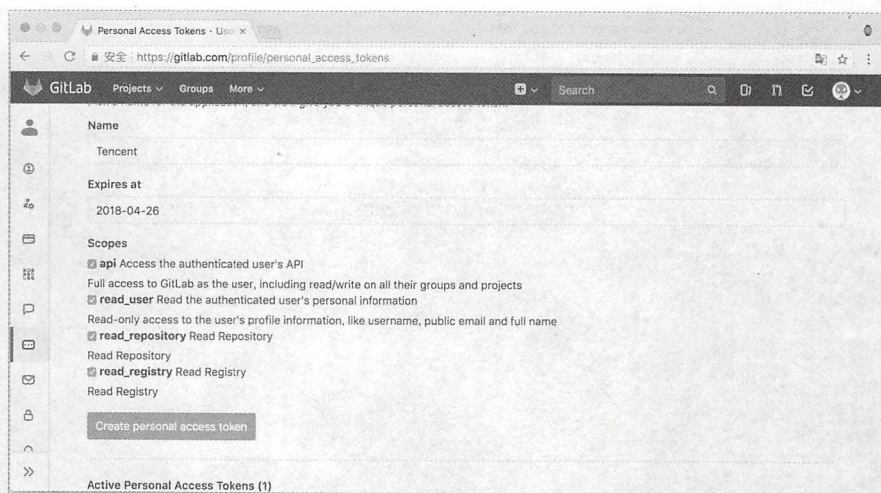


图 6-16

填好之前的页面，然后单击“构建配置”选项，选择要构建的项目，单击“确认”按钮即可，如图 6-17 所示。

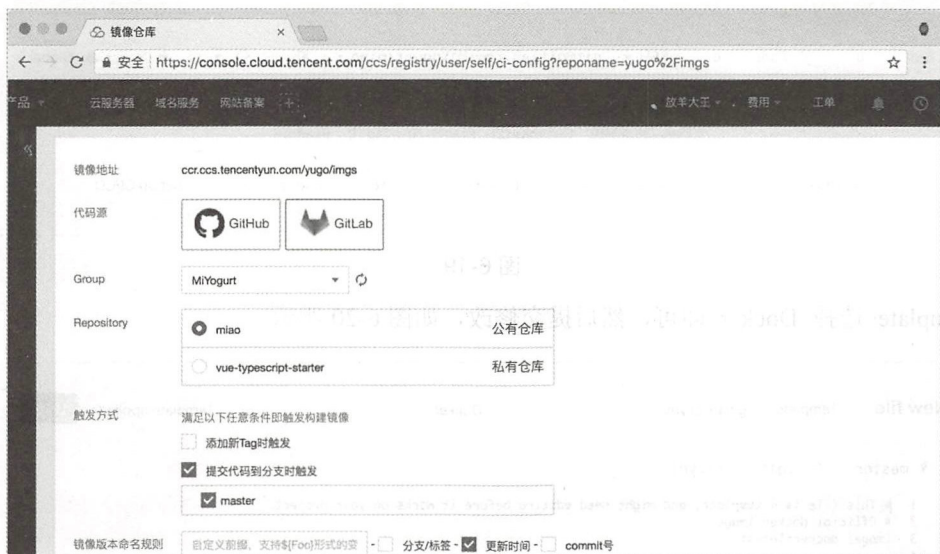


图 6-17

然后单击“立即构建”选项，输入版本号为 v1 即可。
稍等片刻之后，我们便可发现已经构建成功了，如图 6-18 所示。



图 6-18

我们暂时没办法使用触发器，因为没有创建腾讯云提供的容器服务。

6. 创建工作流水线

单击 Set up CI/CD 按钮，如图 6-19 所示。

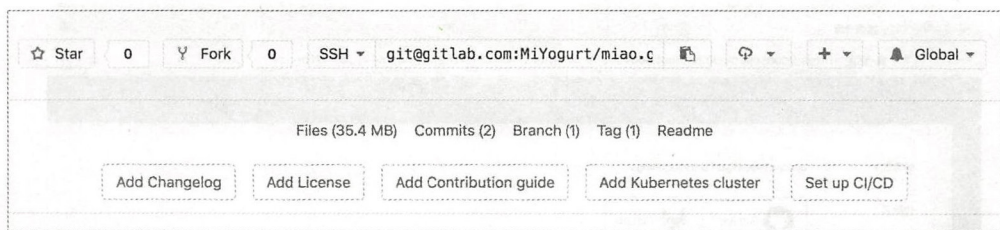


图 6-19

Template 选择 Docker 即可，然后提交修改，如图 6-20 所示。

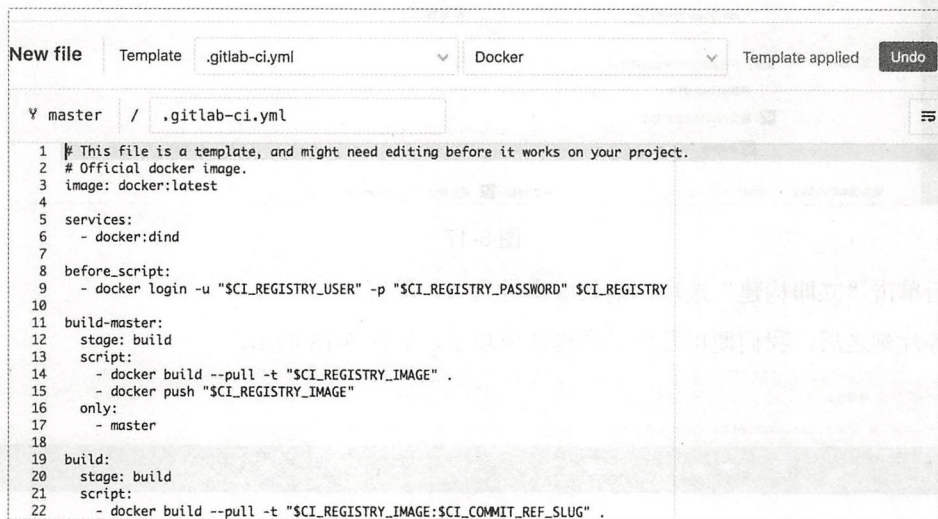


图 6-20

单击侧边栏的 Pipelines 选项可以看到如图 6-21 所示的页面。

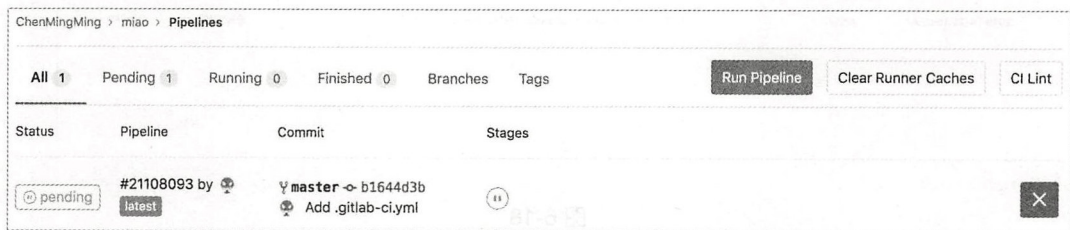


图 6-21

假如一直处于 pending 状态，则说明它在等待 Runner。

GitLab 其实提供了一些 Runner，为了更安全，可以在我们自己的机器上安装 Runner，比如 200 元的树莓派。

安装方法可以在 <https://docs.gitlab.com/runner/install/linux-manually.html> 的文档中找到。下载 gitlab-runner，因为树莓派是 ARM 处理芯片，所以我们要下载 ARM 的版本。

```
sudo wget -O /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-arm
```

添加执行权限：

```
sudo chmod +x /usr/local/bin/gitlab-runner
```

为 Runner 添加一个专门运行的用户：

```
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
```

让该用户可以运行 docker 命令，否则会没有权限：

```
sudo usermod gitlab-runner -a -G docker
```

开始运行：

```
sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
sudo gitlab-runner start
```

在项目设置的 CI/CD 中找到 Runner 设置，我们会看到一段注册的 Token，如图 6-22 所示。

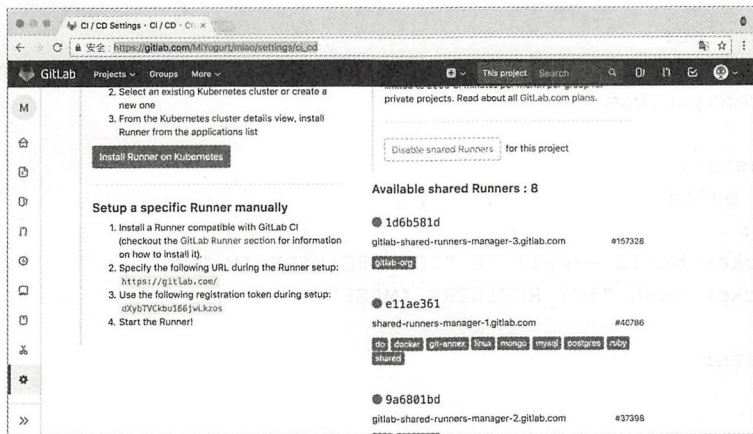


图 6-22

输入 `sudo gitlab-runner register` 进行注册，这里我们使用 `shell` 直接执行，如图 6-23 所示。

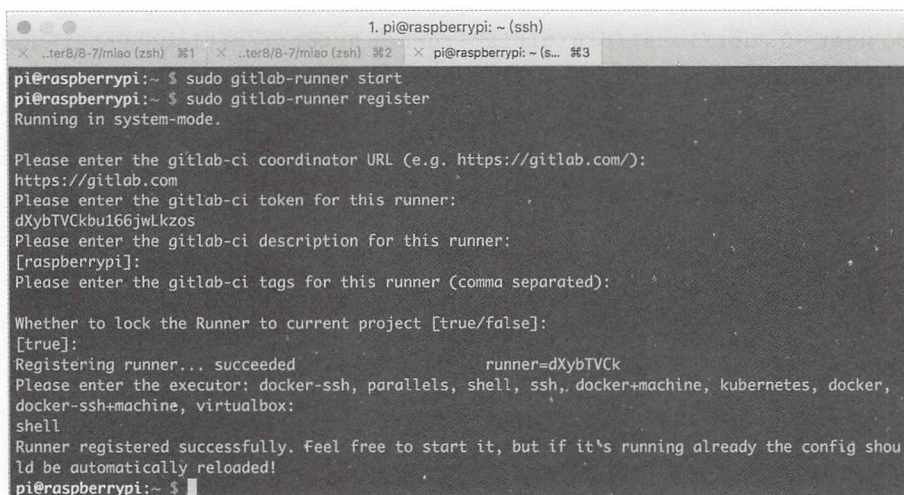


图 6-23

假如状态不是成功的，则可以通过 `sudo gitlab-runner restart` 重新启动。

现在我们把镜像提交到腾讯镜像容器中，修改配置文件如下：

```

variables:
  CI_DEBUG_TRACE: "true"
  CI_REGISTRY_USER: 1733996866
  CI_REGISTRY_PASSWORD: yugo1234
  CI_REGISTRY_IMAGE: ccr.ccs.tencentyun.com/yugo/imgs

before_script:
  - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
    ccr.ccs.tencentyun.com

build-master:
  stage: build
  script:
    - docker build --pull -t "$CI_REGISTRY_IMAGE" .
    - docker push "$CI_REGISTRY_IMAGE"
  only:
    - master

build:
  stage: build

```



```
script:
  - docker build --pull -t "$SCI_REGISTRY_IMAGE:$SCI_COMMIT_REF_SLUG" .
  - docker push "$SCI_REGISTRY_IMAGE:$SCI_COMMIT_REF_SLUG"
except:
  - master
```

根据 Runner 的配置需要等待一些时间，基本都会成功，失败则可能是由于树莓派的存储原因，以及系统架构的问题，比如使用了 x64 CPU 架构的 Docker 镜像。笔者这里只有 16GB 的 SD 卡，所以有的时候会失败，需要手动删除文件，使用真实的主机是没有这些问题的。

7. 部署

当上传成功之后，我们就可以部署了。登录部署服务器，这里直接以本机为例。创建 docker-compose.yml，基本跟之前的脚本一致，只不过应用来自镜像，而不是构建。

```
version: '3.1'
services:
  web:
    image: ccr.ccs.tencentyun.com/yugo/imgs
    restart: always
    depends_on:
      - cache
      - db
    networks:
      - docker_miao
    ports:
      - 7001:7001

  cache:
    image: redis:3.2-alpine
    command: redis-server --appendonly yes --requirepass password
    volumes:
      - egg-redis:/data
    networks:
      - docker_miao

  db:
    image: mysql:5.7
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=password
```



```
adminer:
  image: adminer
  restart: always
  networks:
    - docker_miao
  ports:
    - 8080:8080
```

```
networks:
  docker miao:
```

当然也可以通过 `docker-compose up -d` 在后台运行，如图 6-24 所示。

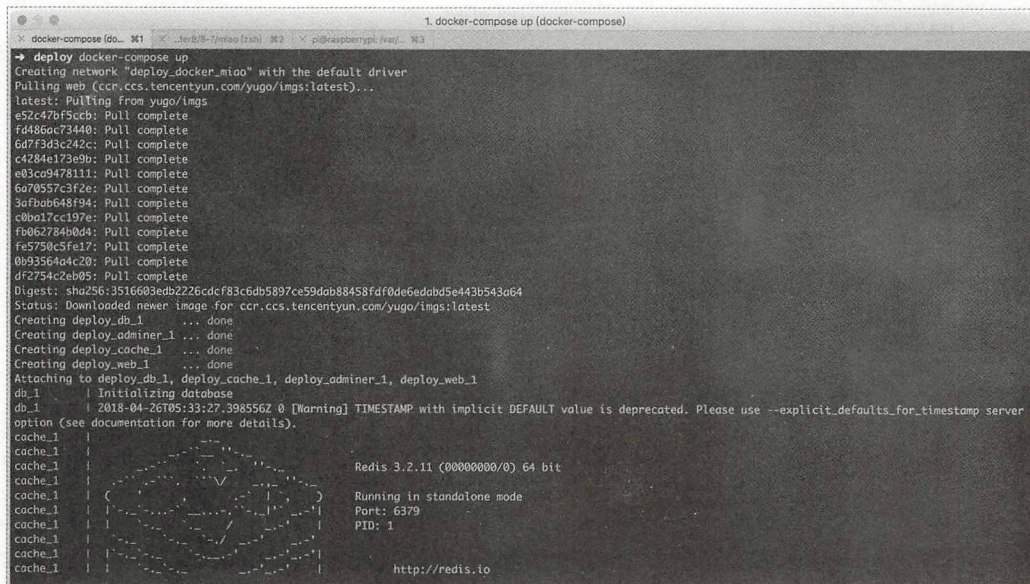


图 6-24

测试管理页面，如图 6-25 所示。

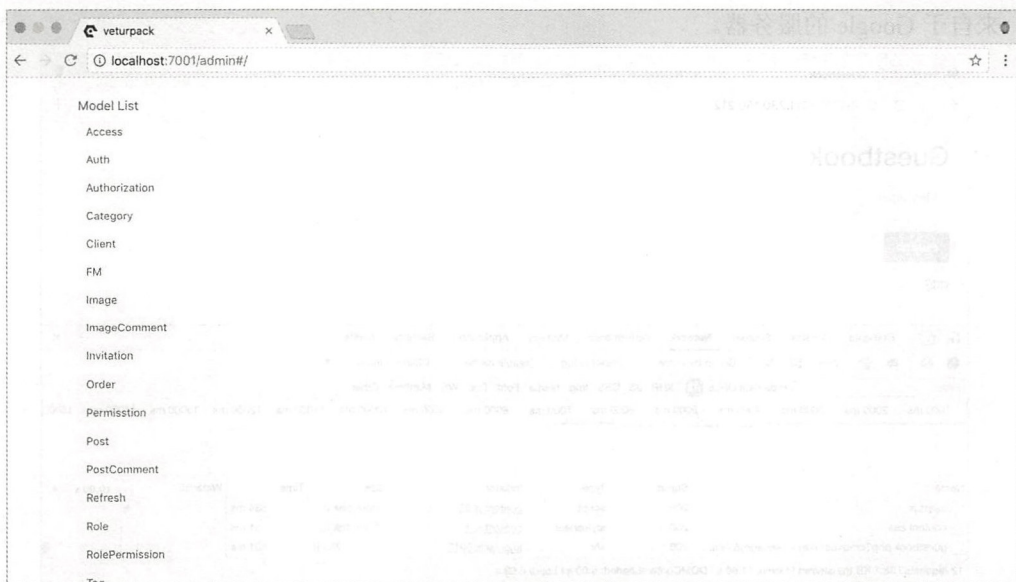


图 6-25

6.8 Kubernetes 集群

Kubernetes 是各大公司都在使用的容器编排技术，其前身是 Google 的 Borg，经过了 Google 多年的实践。

其实常用的技术与组件，云提供商都提供了解决方案，百度云、腾讯云、阿里云都有容器服务。自己搭建 Kubernetes 并不简单，而且维护也是一个问题，所以我们这里使用腾讯云的容器集群来进行部署。

其实这些技术问题都可以通过咨询云服务提供商的客服来解决，毕竟他们的存在就是为了解决这些问题。

笔者也录制过在本地搭建 Kubernetes 开发环境的视频，可以在 <https://nodelover.me/course/k8s-node> 上找到。

6.8.1 简单使用

进入容器服务，单击“免费实验室”选项。

再单击快速实验的“立即创建”按钮。选择 guestbook，单击“确认”按钮，之后我们可

以看到如下的页面，打开服务访问入口，如图 6-26 所示。这将会非常慢，因为它有几个 JS 文件，来自于 Google 的服务器。

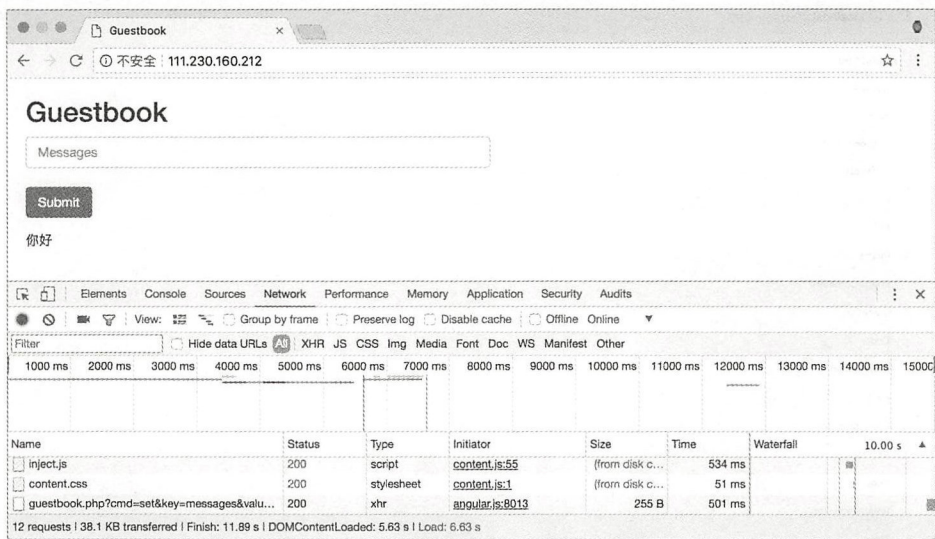


图 6-26

单击左边栏的 Submit 按钮，进入应用中，打开 guestbook，可以发现它有如下几个服务，类型都是 Deployment + Service，如图 6-27 所示。Deployment 可以简单地理解为运行应用的容器，Service 表示容器之间的依赖、网络关系。

基本信息

应用名称	guestbook
运行集群	cls-a4ed48t8
应用描述	guestbook应用,服务访问入口: 111.230.160.212:80
创建时间	2018-04-26 14:25:56
更新时间	2018-04-26 14:25:57

服务列表

服务名	类型	部署状态	运行状态	IP地址①	操作
frontend	Deployment+Service	已部署	运行中	111.230.160.212 ② 172.16.255.106 ③	取消部署 查看Yaml
redis-master	Deployment+Service	已部署	运行中	- 172.16.255.43 ③	取消部署 查看Yaml
redis-slave	Deployment+Service	已部署	运行中	- 172.16.255.239 ③	取消部署 查看Yaml

图 6-27

然后单击左边栏的“服务”选项，查看服务，可以发现 frontend 启动了 2 个实例，如图 6-28 所示。

名称①	监控	日志	状态	运行/预期数量	IP地址①	负载均衡	标签(label) ▾	创建时间 s	操作
redis-slave	山	图	运行中	1/1个	172.16.255.239	未启用	qcloud-appre...	2018-04-26 14:25...	更新实例数量 更新服务 更多 +
frontend	山	图	运行中	2/2个	111.230.160.212 172.16.255.106	lb-cin2bq9b	qcloud-app:fr...	2018-04-26 14:25...	更新实例数量 更新服务 更多 +
redis-master	山	图	运行中	1/1个	172.16.255.43	未启用	qcloud-appre...	2018-04-26 14:25...	更新实例数量 更新服务 更多 +

图 6-28

6.8.2 如何创建应用

单击左边栏的 Submit 按钮，选择“更新应用”选项，然后单击“下一步”按钮，如图 6-29 所示。

新建					请输入应用名称
应用名称	部署状态	服务状态	创建时间	操作	
guestbook	已部署:3个 未部署:0个	正常	2018-04-26 14:25:56	更新应用 部署 删除	

图 6-29

我们可以看到这样的页面，这些服务的定义其实都是一些 yaml 定义文件，它的内部会把这些 yaml 格式的文件转换成请求发送给 Kubernetes 的服务端，然后创建服务。Kubernetes 是基于 Go 语言开发的，花括号加点引用变量是 Go 语言模板的标准形式。

- apiVersion 是本文件发送请求的版本，就像我们之前后端服务写的/api/v1、/api/v2 一样；
- kind 表示类型；
- metadata 标识本服务的原数据，比如叫什么名字，在哪个命名空间下，命名空间是用于保证不冲突；
- spec 是对本服务信息的描述，比如启动几个实例，使用什么镜像构建，对资源访问限制大小等。不同类型的 spec 不同，比如 kind 为 Deployment 和 Service；
- --- 表示文件分割线。

kind 类型为 Service 的 spec 描述主要是一些网络类型、端口映射的配置信息，如图 6-30 所示。

所有的配置项都可以在文档中找到，不懂这些其实也没大关系。

把这几个服务先删除，我们来创建一个公网可以访问的 MySQL 服务。单击“从 UI 导入服务”选项之后，单击“选择镜像”按钮，如图 6-31 所示。



图 6-30

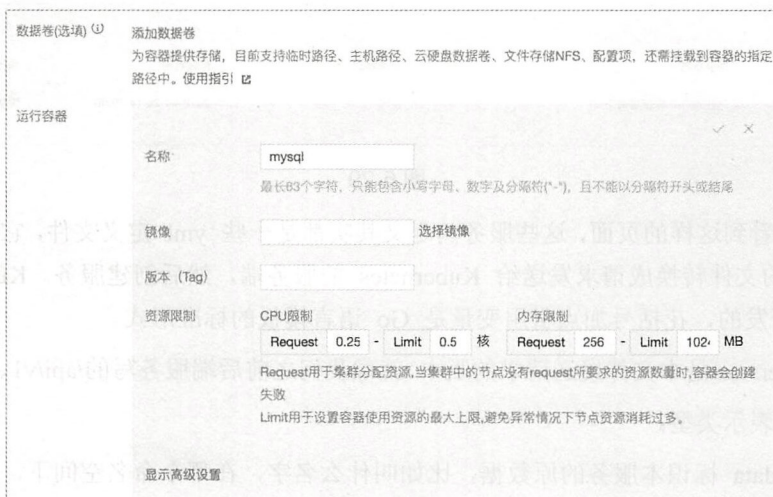


图 6-31

从 DockerHub 中选择 MySQL 镜像，然后选择 5.7 版本，如图 6-32 所示。



图 6-32

单击“高级设置”选项，添加环境变量 `MYSQL_ROOT_PASSWORD`，如图 6-33 所示。

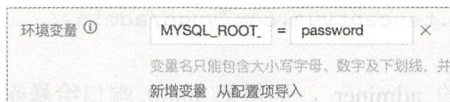


图 6-33

添加 3306 端口，把 3306 端口暴露出去，图 6-34 所示。



图 6-34

然后我们可以看到生成的 yml 文件，如图 6-35 所示。



图 6-35

由于腾讯云只提供了 DockerHub 官方的镜像，我们自己把下载的 `adminer` 镜像推送到腾讯云容器仓库中。

```
docker tag adminer ccr.ccs.tencentyun.com/yugo/adminer
docker push ccr.ccs.tencentyun.com/yugo/adminer
```

选择我们刚刚推送上去的 adminer，记得把 8080 端口给暴露出来，如图 6-36 所示。



图 6-36

完成之后，记得把 namespace 修改为自己的值，如图 6-37 所示。

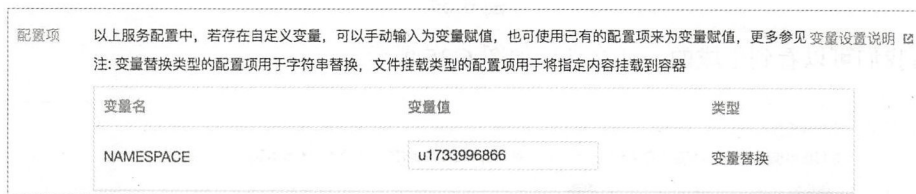


图 6-37

选择操作中的“部署”选项，删除之前的部分，部署当前更新好的部分。

adminer 会启动失败，事件错误告诉我们资源不够，如图 6-38 所示。



图 6-38

修改 yml 的配置文件, 修改 MySQL 的资源配置, requests 表示最低配置要求, limits 表示最高配置上限, 如图 6-39 所示。

服务名	操作	内容
mysql	删除	22 imagePullPolicy: Always
adminer	删除	23 env:
		24 - name: MYSQL_ROOT_PASSWORD
		25 value: password
		26 name: mysql
		27 resources:
		28 limits:
		29 cpu: 300m
		30 memory: 400Mi
		31 requests:
		32 cpu: 250m
		33 memory: 256Mi
		34 securityContext:
		35 privileged: false
		36 serviceAccountName: ""
		37 volumes: null

图 6-39

修改 adminer 的资源配置, 如图 6-40 所示。

选择应用模板后, 将从模板内容复制到当前应用, 继续编写或修改当前应用不会影响模板本身。		
服务名	操作	内容
mysql	删除	20 containers:
adminer	删除	21 - image: ccr.ccs.tencentyun.com/yugo/admin:latest
		22 imagePullPolicy: Always
		23 name: adminer
		24 resources:
		25 limits:
		26 cpu: 500m
		27 memory: 1Gi
		28 requests:
		29 cpu: 250m
		30 memory: 720Mi
		31 securityContext:
		32 privileged: false
		33 serviceAccountName: ""
		34 volumes: null
		35 status: {}
		36 ---
		37 apiVersion: v1
		38 kind: Service
新增空服务 从UI导入服务		

图 6-40

从服务列表中看到, 服务都已经运行成功了, 如图 6-41 所示。

adminer	运行中	1/1个	111.230.207.121 172.16.255.164	lb-18zdbx4x	qcloud-appa...	2018-04-26 16:20...	更新实例数量 更新服务 更多
mysql	运行中	1/1个	193.112.232.124 172.16.255.60	lb-4aovhz29	qcloud-appm...	2018-04-26 16:13...	更新实例数量 更新服务 更多

图 6-41

按照提供的信息填写表单, 并进行登录, 如图 6-42 所示。

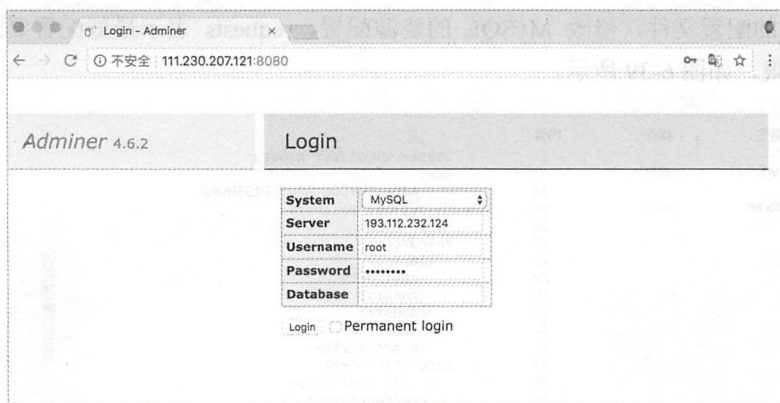


图 6-42

可以看到登录成功了，如图 6-43 所示。

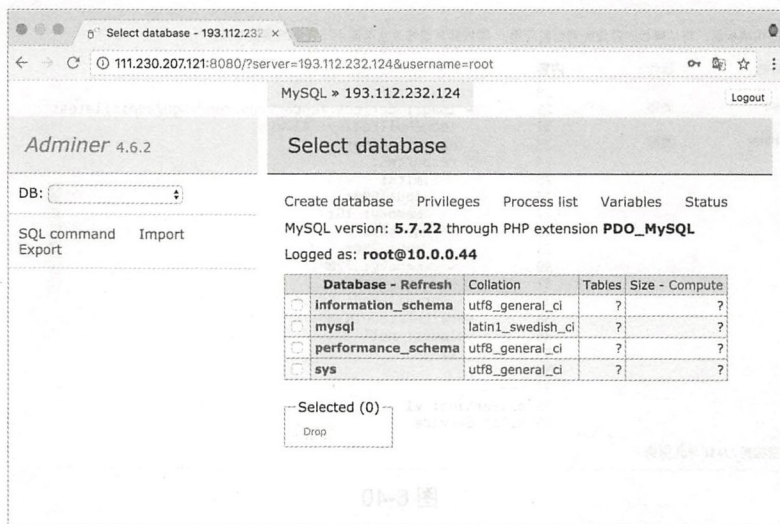


图 6-43

6.8.3 命令行管理

1. 下载 kompose

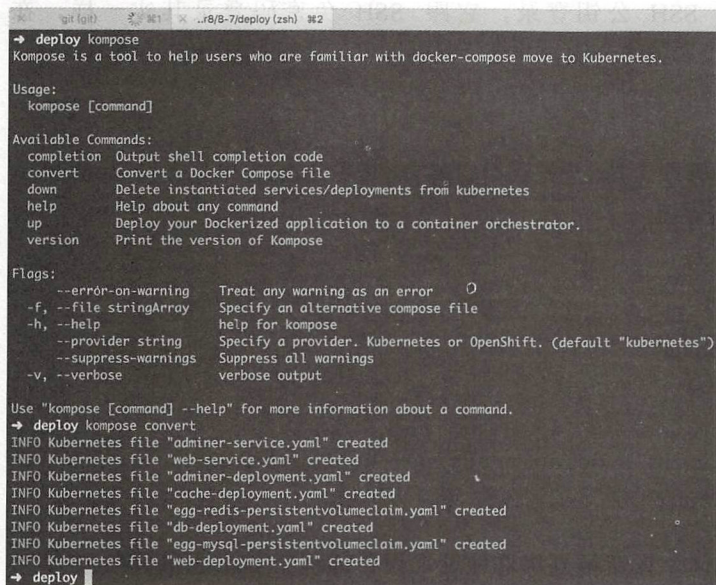
kompose 是将 docker-compose.yml 配置文件转化为 kubernetes 配置文件的工具。

```
brew install kompose
```

2. 转换文件

来到上一节部署的 `docker-compose.yml` 所在的文件夹，执行转换命令，如图 6-44 所示。

```
kompose convert
```



```
→ deploy kompose
Kompose is a tool to help users who are familiar with docker-compose move to Kubernetes.

Usage:
  kompose [command]

Available Commands:
  completion  Output shell completion code
  convert      Convert a Docker Compose file
  down        Delete instantiated services/deployments from Kubernetes
  help        Help about any command
  up          Deploy your Dockerized application to a container orchestrator.
  version     Print the version of Kompose

Flags:
  --error-on-warning  Treat any warning as an error
  -f, --file stringArray  Specify an alternative compose file
  -h, --help            help for kompose
  --provider string    Specify a provider. Kubernetes or OpenShift. (default "kubernetes")
  --suppress-warnings  Suppress all warnings
  -v, --verbose         verbose output

Use "kompose [command] --help" for more information about a command.
→ deploy kompose convert
INFO Kubernetes file "adminer-service.yaml" created
INFO Kubernetes file "web-service.yaml" created
INFO Kubernetes file "adminer-deployment.yaml" created
INFO Kubernetes file "cache-deployment.yaml" created
INFO Kubernetes file "egg-redis-persistentvolumeclaim.yaml" created
INFO Kubernetes file "db-deployment.yaml" created
INFO Kubernetes file "egg-mysql-persistentvolumeclaim.yaml" created
INFO Kubernetes file "web-deployment.yaml" created
→ deploy
```

图 6-44

会看到我们有如图 6-45 所示的几个文件，以 `persistentvolumeclaim` 结尾的是容器挂载卷的配置，基本每一个服务都有 `deployment` 和 `service`。

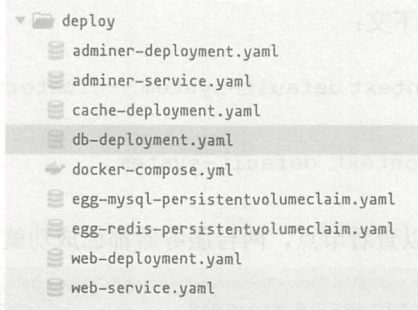


图 6-45

3. 创建集群

免费的配置肯定是不够的，所以我们要自己创建集群。创建的时候要注意集群网络，CIDR

表示子网掩码可以借位，它告诉你是 16 位的子网掩码，10.0.x.x 的前两位为网络位，x 表示任意不超过 255 的值，网络位相同的计算机才可以进行正常通信（无路由转发的情况下，同时也表示在同一个子网里），新建的时候也要保持一致，默认是 24 位的。同时网路的区域也要保证一致。

然后我们选择 2 核心、4GB 内存的配置，选择 CentOS 系统，CentOS 相对而言比 ubuntu 稳定些。为了安全，我们使用 SSH 公钥登录，它跟 SSH 免密码登录其实一样，在 ~/.ssh/id_rsa.pub 上通过 cat 命令查看你的公钥，并进行复制即可，还可以打开自动调节。

4. 安装管理工具

kubectl 是 Kubernetes 集群的管理工具，首先我们安装它，其实 Linux 也是支持 homebrew 的，具体内容请查看 <http://linuxbrew.sh>。

```
brew install kubectl
```

在集群信息中可以查看集群凭证。

设置用户凭证：

```
kubectl config set-credentials default-admin --username=admin --password=
BzJfhkNXd0KzfNdWaf0qE03E4NkzunxV
```

单击凭证里的“开启外网访问”按钮得到集群地址，设置远程集群地址和 HTTPS 连接证书。

```
kubectl config set-cluster default-cluster --server=https://cls-nl74wshi.
ccs.tencent-cloud.com --certificate-authority=./cluster-ca.crt
```

创建上下文，并使用该上下文：

```
kubectl config set-context default-system --cluster=default-cluster --user=
default-admin
```

```
kubectl config use-context default-system
```

运行 `kubectl get node` 可以查看节点，两台服务器都已成功就绪，如图 6-46 所示。

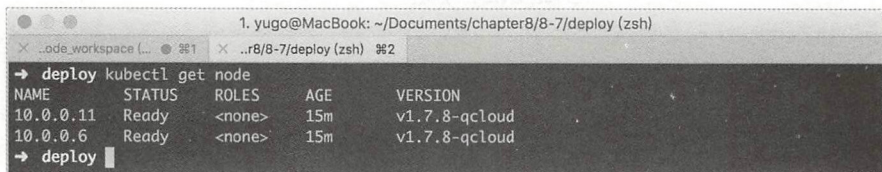


图 6-46

5. 尝试 Kompose

运行 `kompose up` 启动应用（不建议读者尝试，`persistenvolumeclam` 会创建云硬盘，这里创建了两个，运行一次就是 6 元钱，笔者测试的时候创建了 9 个，联系过客服，被告知是自己的问题，无法退还，这让人有点不爽），这个命令使用的其实是 `docker-compose.yml` 文件，通过 `convert` 生成的，需要通过 `kubectl create -f` 手动创建服务，如图 6-47 所示。

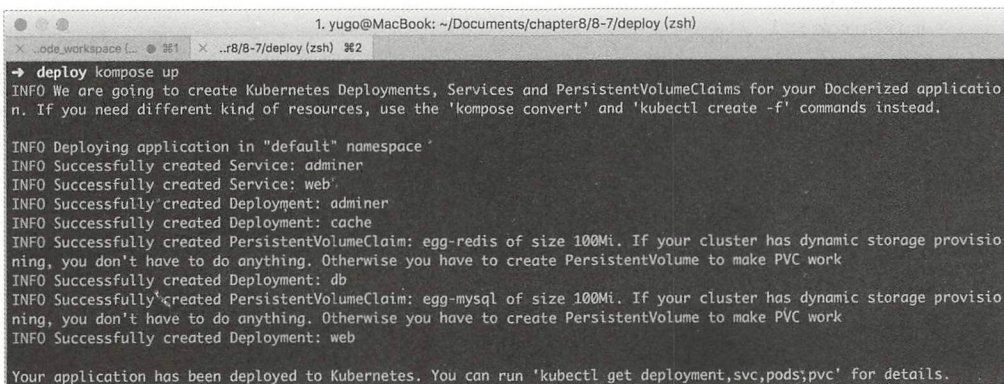
A terminal window titled '1. yugo@MacBook: ~/Documents/chapter8/8-7/deploy (zsh)' showing the output of the 'kompose up' command. The output indicates that services 'adminer', 'web', 'cache', 'db', and 'mysql' were successfully created as Kubernetes Deployments and PersistentVolumeClaims in the 'default' namespace. It also provides instructions on how to check the deployment details using 'kubectl'.

图 6-47

但是 `db` 容器和 `Web` 容器启动失败，还是因为容器卷挂载的问题，以及镜像拉取的原因。通过 `kompose` 创建的服务无法在 `UI` 界面上做更改，所以我们没办法方便地做修改，我们尽量规避写部署 `yml` 的脚本，所以还是使用 `UI` 的应用进行创建。

删除之前的服务：

```
kompose down
```

6.8.4 通过 UI 创建应用

创建的步骤就不细说了，笔者把生成的 `yml` 展示出来，大家自行对比，不要选择云硬盘，每选择一次它就会帮你创建一个云硬盘，也就是 3 块钱。

- `db`

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    description: db
  creationTimestamp: null
  name: db
  namespace: '{{.NAMESPACE}}'
```



```
spec:
  replicas: 1
  revisionHistoryLimit: 5
  selector: {}
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
      - env:
        - name: MYSQL_DATABASE
          value: miao
        - name: MYSQL_ROOT_PASSWORD
          value: password
        image: mysql:5.6
        imagePullPolicy: Always
        name: db
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
        requests:
          cpu: 250m
          memory: 256Mi
      securityContext:
        privileged: false
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: db
    serviceAccountName: ""
  volumes:
  - hostPath:
      path: /data/mysql
```

```
      name: db
status: {}

---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  name: db
  namespace: '{{.NAMESPACE}}'
spec:
  ports:
    - name: tcp-3306-3306-xczub
      nodePort: 0
      port: 3306
      protocol: TCP
      targetPort: 3306
  selector: {}
  type: ClusterIP
status:
  loadBalancer: {}
```

- miao

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  creationTimestamp: null
  name: miao
  namespace: '{{.NAMESPACE}}'
spec:
  replicas: 1
  revisionHistoryLimit: 5
  selector: {}
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
```

```
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
        - image: ccr.ccs.tencentyun.com/yugo/imgs:latest
          imagePullPolicy: Always
          name: web
          resources:
            limits:
              cpu: 500m
              memory: 1Gi
            requests:
              cpu: 250m
              memory: 256Mi
          securityContext:
            privileged: false
          serviceAccountName: ""
          volumes: null
      status: {}

---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  name: miao
  namespace: '{{.NAMESPACE}}'
spec:
  ports:
    - name: tcp-7001-7001-o8dcf
      nodePort: 0
      port: 7001
      protocol: TCP
      targetPort: 7001
  selector: {}
  type: LoadBalancer
```



```
status:
  loadBalancer: {}
```

- cache

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  creationTimestamp: null
  name: cache
  namespace: '{{.NAMESPACE}}'
spec:
  replicas: 1
  revisionHistoryLimit: 5
  selector: {}
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
    spec:
      containers:
      - args:
        - redis-server
        - --appendonly
        - "yes"
        - --requirepass
        - password
        image: redis:3.2-alpine
        imagePullPolicy: Always
        name: cache
      resources:
        limits:
          cpu: 500m
          memory: 1Gi
```




```
    requests:
      cpu: 250m
      memory: 256Mi
    securityContext:
      privileged: false
    volumeMounts:
      - mountPath: /data
        name: redis
    serviceAccountName: ""
    volumes:
      - hostPath:
          path: /data/redis
          name: redis
status: {}

---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  name: cache
  namespace: '{{.NAMESPACE}}'
spec:
  ports:
    - name: tcp-6379-6379-zez15
      nodePort: 0
      port: 6379
      protocol: TCP
      targetPort: 6379
  selector: {}
  type: ClusterIP
status:
  loadBalancer: {}
```

现在已经可以正常访问了，但是由于拉取表结构的接口还是之前的 `localhost:7001`，需要重新编译后端的源码才能解决，如图 6-48 所示。

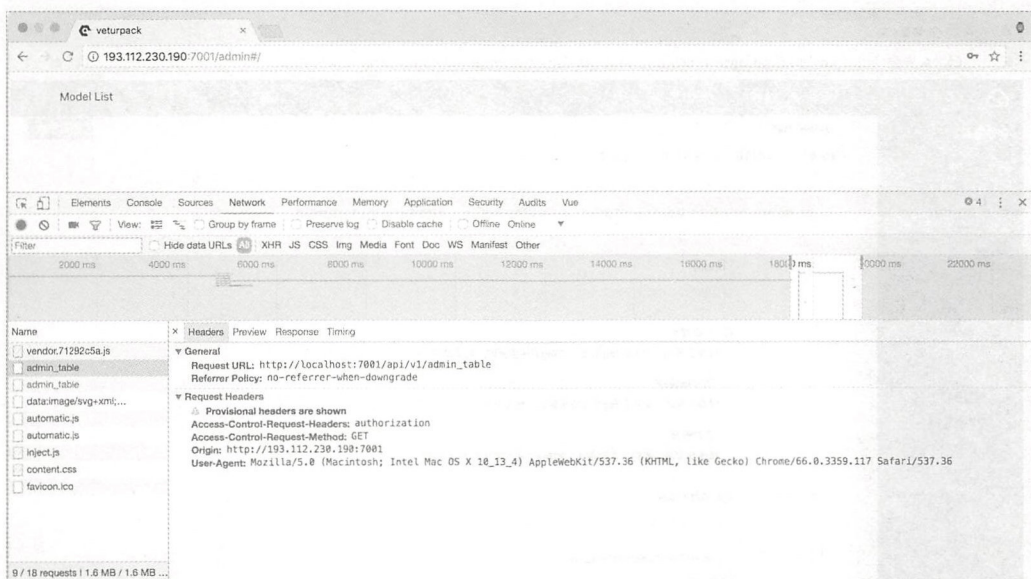


图 6-48

接口是没有任何问题的，如图 6-49 所示。

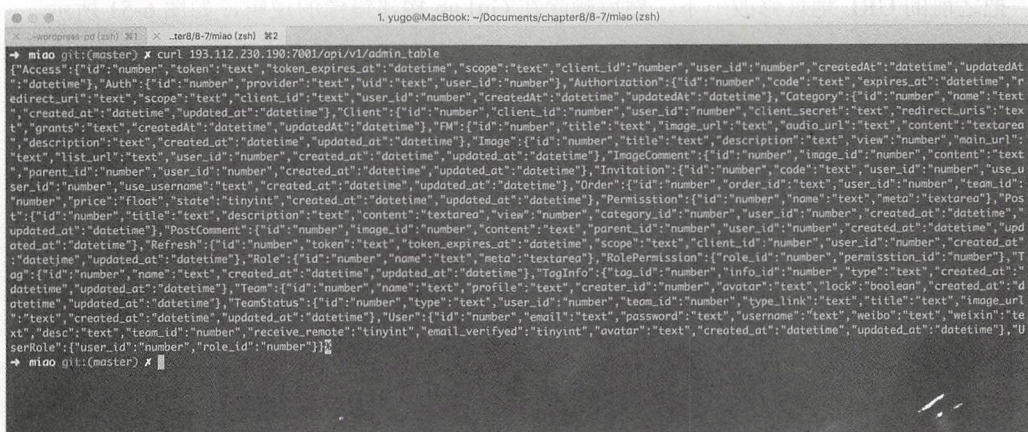


图 6-49

6.8.5 添加持续集成

添加一个触发器，记得要保证你授权给了腾讯云，如图 6-50 所示。



图 6-50

把之前的 URL 地址修改一下，然后删除在 GitLab 构建镜像的逻辑，如图 6-51 所示。

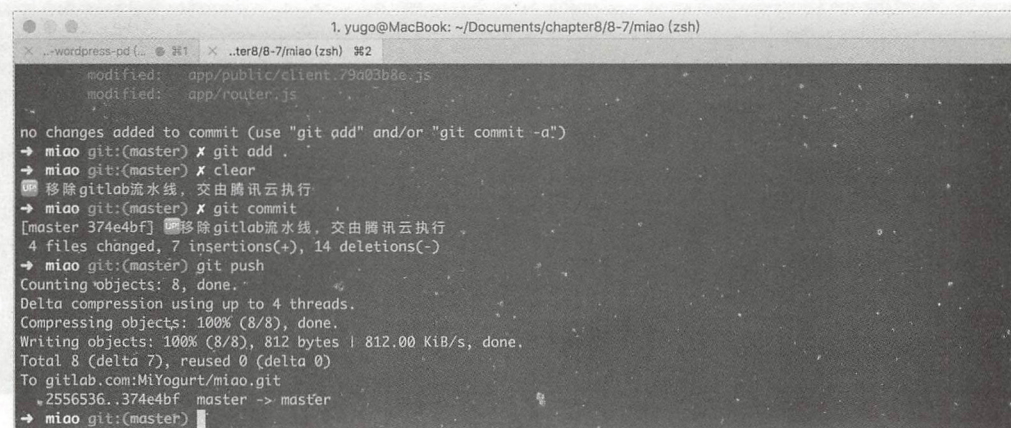


图 6-51

现在已经构建成功了，并成功地启动了容器，如图 6-52 所示。

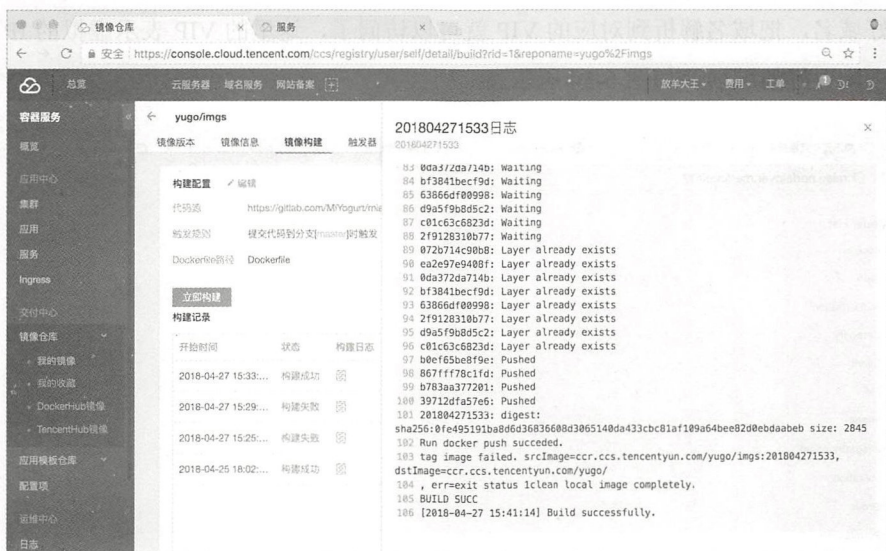


图 6-52

6.8.6 固定 IP 地址

默认的公网访问其实也是分配的，需要付费，现在创建一个 ingress 实例。

然后进行转发配置，它可以把请求转发到内部网络的某一个服务中，如图 6-53 所示。

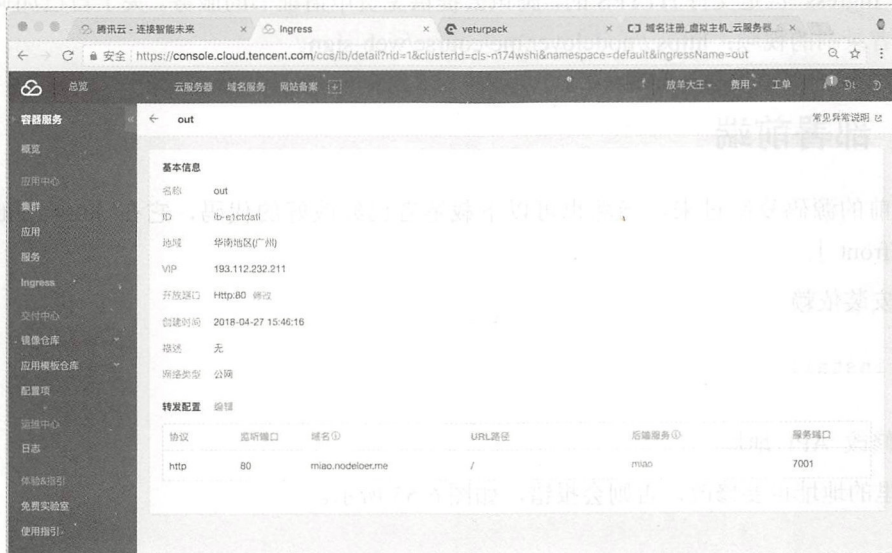


图 6-53





购买好域名，把域名解析到对应的 VIP 就可以访问了，这里的 VIP 表示虚拟的 IP 地址。访问刚配置好的域名，如图 6-54 所示。

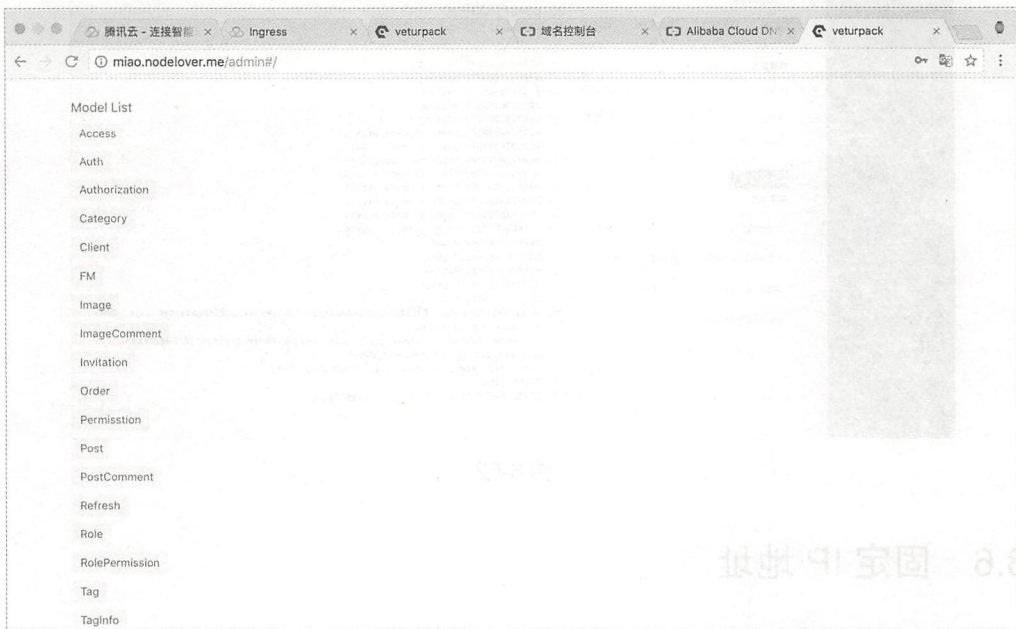


图 6-54

当然 Ingress 也是支持 HTTPS 的，腾讯云提供免费申请证书的服务。关于自己如何验证可以查看笔者录制的视频：<https://nodelover.me/course/web-sign>。

6.8.7 部署前端

将之前的源码复制过来，当然也可以下载笔者已经改好的代码，它在 <https://gitlab.com/MiYogurt/front> 上。

- 安装依赖

```
npm install
```

- 修改 API 地址

API 里的地址也要修改，否则会报错，如图 6-55 所示。



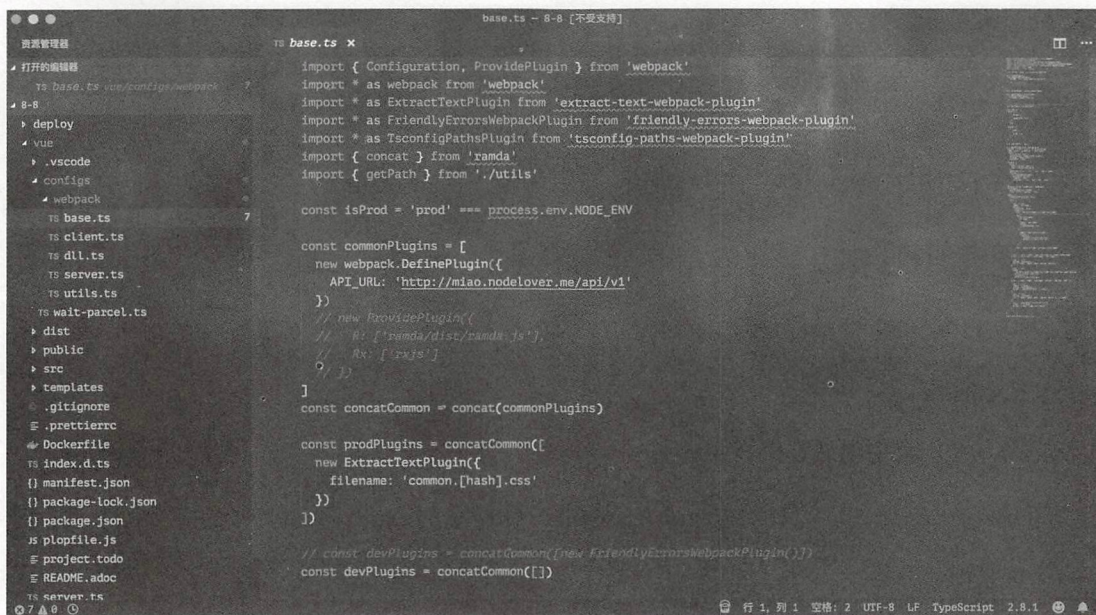


图 6-55

- 编译前端

```
npm run build:client
```

- 修改 server.ts

在生产环境中不需要的模块要进行删除，通过 isProd 判断一下，具体代码查看仓库源码。

- 编译后端

```
npm run build:server
```

```
tsc server.ts
```

- 添加 Dockerfile 文件

```
FROM node:9.2.0
```

```
RUN ["mkdir", "-p", "/usr/src/application"]
```

```
WORKDIR /usr/src/application
```

```
COPY package.json /usr/src/application/
```





```
RUN ["npm", "install", "--production", "--registry=https://registry.npm.taobao.org"]
```

```
COPY . /usr/src/application
```

```
EXPOSE 4000
```

```
CMD NODE_ENV=prod node server.js
```

- 构建镜像

记得在控制台中创建这个仓库。

```
docker build -t ccr.ccs.tencentyun.com/yugo/front .
```

- 推送

```
docker push ccr.ccs.tencentyun.com/yugo/front
```

- 修改 db 的网络类型

修改为 LoadBalancer，这样我们会得到一个公网的 IP 地址，这样才可以从本地连接数据库，当然也可以启动一个 adminer 服务。

db	运行中	1/1个	193.112.23... 172.16.255...	lb-5m2m...	qcloud-ap.
----	-----	------	--------------------------------	------------	------------

- 导入数据

用 SequelPro 连接数据库，将 SQL 拖到 Query 的窗口，选择所有 SQL 运行即可，如图 6-56 和图 6-57 所示。



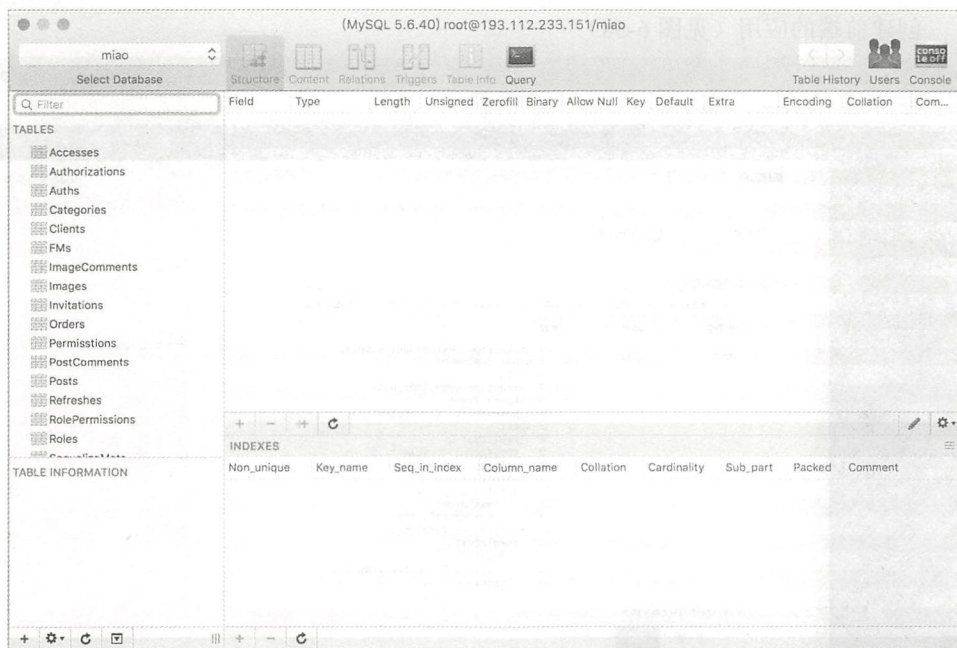


图 6-56

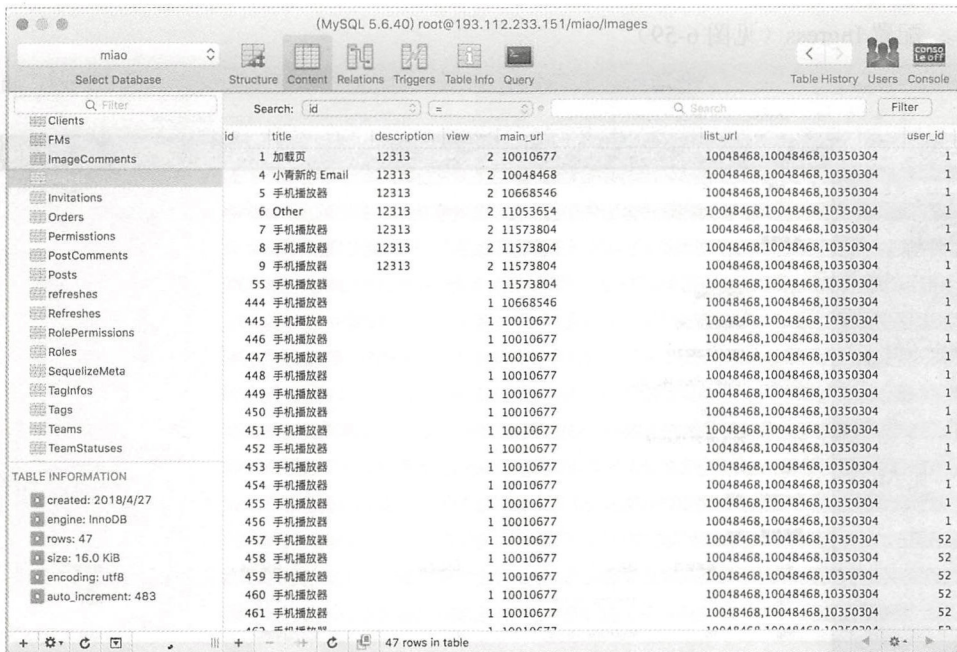


图 6-57





- 创建前端的应用（见图 6-58）



图 6-58

- 配置 Ingress（见图 6-59）

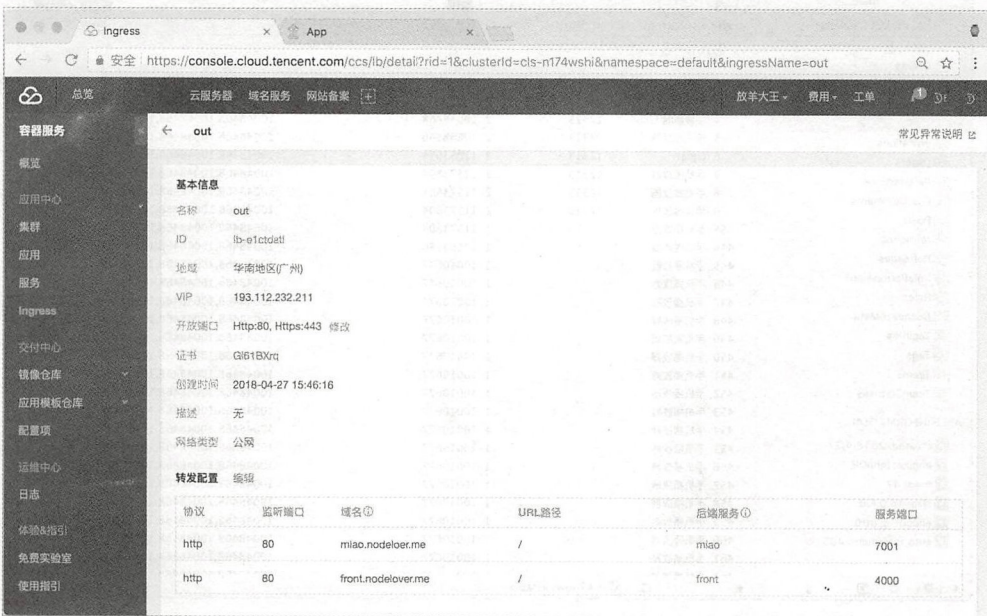


图 6-59



- 访问页面

此时还会有一些后台的跨域白名单没有配置，所以前端载入图片还是会发生错误，配置一下即可。



图 7-1



7 chapter

第 7 章 性能分析与优化

7.1 服务器性能分析与测试

1. 安装

进入 <https://www.aliyun.com/product/nodejs>，登录之后直接申请开通即可，服务是免费的。

- 创建一个新应用，输入名称，会得到如图 7-1 所示的信息。



图 7-1

2. 使用 Egg 进行安装

我们用一个新的项目进行测试。

```
egg-init --type=simple egg-test
```



```
npm i nodeinstall -g
cd egg-test
nodeinstall --install-alinode ^3
```

nodeinstall 会安装运行环境，性能分析工具对 node 做了修改，所以我们要安装它的运行环境。

```
npm i egg-alinode -S
```

安装完组件之后，开启插件，并配置 ID 与 secret:

```
exports.alinode = {
  enable: true,
  package: 'egg-alinode'
}
config.alinode = {
  server: 'wss://agentserver.node.aliyun.com:8080',
  appid: '43688',
  secret: '213599ab5abc6c6d0353980eb1f7fffd26c5f552'
}
```

3. 查看表盘

- 启动应用

```
npm run start
```

然后回到控制台，等待几分钟就可以看到表盘了，如图 7-2 所示。由于笔者是在本地计算机启动的，所以有些数据可能没有获取。



图 7-2

4. 普通安装

这里笔者准备了一台新的云服务器进行测试。



- 下载安装脚本

```
wget -q https://raw.githubusercontent.com/aliyun-node/alinode-all-in-one/master/alinode_all.sh
```

- 以交互的方式启动脚本

```
bash -i alinode_all.sh
```

按照提示安装即可，输入你的 ID 与 secret，这里全部选择默认的。

```
nohup agenthub yourconfig.json &
```

这样我们就以守护进程的方式启动了脚本，它会在我们运行 node 应用的时候将信息发送到阿里云的服务器上进行统计。

这里笔者默认安装的 node 版本是 v10.1.0:

```
[root@VM_138_125_centos ~]# node -v  
v10.1.0
```

- 初始化 Egg

```
npm i -g egg-init  
egg-init --type simple start  
cd start && npm install
```

- 启动

由于版本有些新，会有一个警告导致启动失败，使用 ignore-stderr 忽略错误即可。

```
ExperimentalWarning: The fs.promises API is experimental  
npm run start -- --ignore-stderr
```

5. 修复错误

当我们检查运行进程的时候发现有一个有问题，需要修改一下 youconfig.json，如图 7-3 所示。



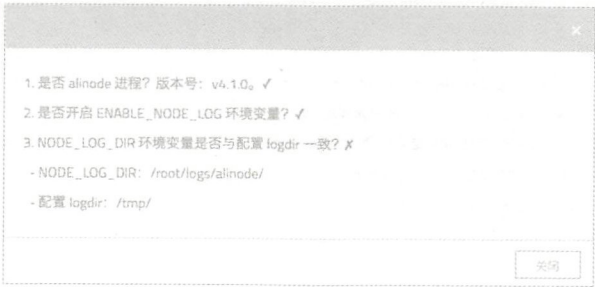


图 7-3

把 logdir 修改为上面的路径即可，如图 7-4 所示。

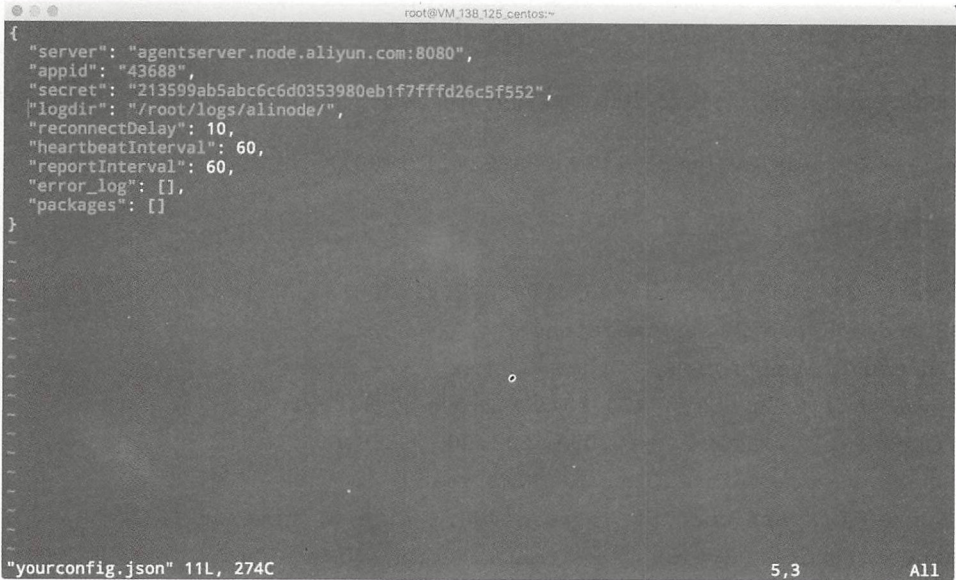


图 7-4

- 重启 agenthub

```
agenthub stop all
nohup agenthub yourconfig.json &
```

现在就一致了，如图 7-5 所示。

6. 压力测试

压测的工具很多，下面介绍几个 Node.js 相关的。



图 7-5

Wrk

```
npm i -g wrk
```

运行命令进行压测。它代表 200 个线程去请求地址，请求时长为 120 秒，即 2 分钟。

```
wrk -c 200 -d 120 http://139.199.227.41:7001/
```

得到以下数据：

```
Running 2m test @ http://139.199.227.41:7001/
 2 threads and 200 connections
Thread Stats   Avg      Stdev   Max    +/- Stdev
  Latency  372.22ms  450.15ms  2.00s   83.97%
  Req/Sec  144.10    40.03   683.00   76.60%
34468 requests in 2.00m, 11.97MB read
Socket errors: connect 0, read 790, write 0, timeout 1683
Requests/sec:  286.99
Transfer/sec:   102.02KB
```

项目名称与说明如表 7-1 所示。

表 7-1

项 目	名 称	说 明
Avg	平均值	每次测试的平均值
Stdev	标准偏差	结果的离散程度，越高说明越不稳定
Max	最大值	最大的一次结果
+/- Stdev	正负一个标准差占比	结果的离散程度，越大越不稳定
Requests/sec	每秒处理请求数	
Transfer/sec	每秒传输数	

Autocannon

```
autocannon -c 200 -d 120 http://139.199.227.41:7001/
```

得到的结果跟 wrk 的基本差不多。

Stat	Avg	Stdev	Max
Latency (ms)	618.18	1387.12	9623.96
Req/Sec	286.99	21.52	517
Bytes/Sec	104 kB	7.5 kB	188 kB

34k requests in 120s, 12.5 MB read

231 errors (231 timeouts)

当我们进行压测的时候，其实检测平台会显示有一些性能消耗，如图 7-6 所示。



图 7-6

这是笔者测试百度的结果，当然这些指标只是一部分，具体看哪些指标，下面会继续分析。

```
autocannon -c 200 -d 120 https://www.baidu.com
Running 120s test @ https://www.baidu.com
200 connections
```

Stat	Avg	Stdev	Max
Latency (ms)	348.8	312.8	8652.17


```
Req/Sec      571.4   41.12  620
Bytes/Sec     8.85 MB 652 kB 9.59 MB
```

```
69k requests in 120s, 1.06 GB read
1 errors (1 timeouts)
```

locust

这是一个 Python 的工具，比较强大，文档在 <https://locust.io> 可以找到。

```
pip install locustio
```

创建 index.py 文件，记得修改 host 为你自己的地址。

```
from locust import HttpLocust, TaskSet, task
```

```
def index(l):
```

```
    l.client.get("/")
```

```
class UserTasks(TaskSet):
```

```
    # one can specify tasks like this
```

```
    tasks = [index]
```

```
    # but it might be convenient to use the @task decorator
```

```
    @task
```

```
    def page404(self):
```

```
        self.client.get("/does_not_exist")
```

```
class WebsiteUser(HttpLocust):
```

```
    """
```

```
    Locust user class that does requests to the locust web server running
on localhost
```

```
    """
```

```
    host = "http://139.199.227.41:7001"
```

```
    min_wait = 2000
```

```
    max_wait = 5000
```

```
    task_set = UserTasks
```

运行命令：

```
locust -f index.py
```

打开 <http://localhost:8089>, 输入 200 与 2, 代表 200 个用户, 每个用户 1 秒大概有 2 个请求。

大概每秒能处理 26 个请求, 响应时间最高达 89 ms, 如图 7-7 所示, 效果比较好, 因为网页比较简单。



图 7-7

现在我们来重新测试一下, 由于应用太简单了, 看不出实际效果, 这里启动一个 nodelover.me 的本地服务, 也可以启动一个之前写好的前端服务。

修改 `index.py`, 任意添加几个页面:

```
from locust import HttpLocust, TaskSet, task

def index(l):
    l.client.get("/")

class UserTasks(TaskSet):
```

```
# one can specify tasks like this
tasks = [index]

# but it might be convenient to use the @task decorator
@task
def pagelogin(self):
    self.client.get("/login")
@task
def courses(self):
    self.client.get("/courses")

class WebsiteUser(HttpLocust):
    """
    Locust user class that does requests to the locust web server running
on localhost
    """
    host = "http://localhost:7001"
    min_wait = 2000
    max_wait = 5000
    task_set = UserTasks
locust -f index.py
```

参数修改为 500 个用户和每秒 4 个请求。

打开活动监视器，可以看到 node 的 CPU 使用率已经超过了 100%，如图 7-8 所示。内存维持在 250MB 左右，因为不存在内存泄漏。

进程名称	% CPU	CPU 时间	线程	闲置唤醒	PID	用户
node	104.2	2:06.27	10	11	16856	Yugo

图 7-8

然后我们来查看结果图，图 7-9 是总体的趋势。

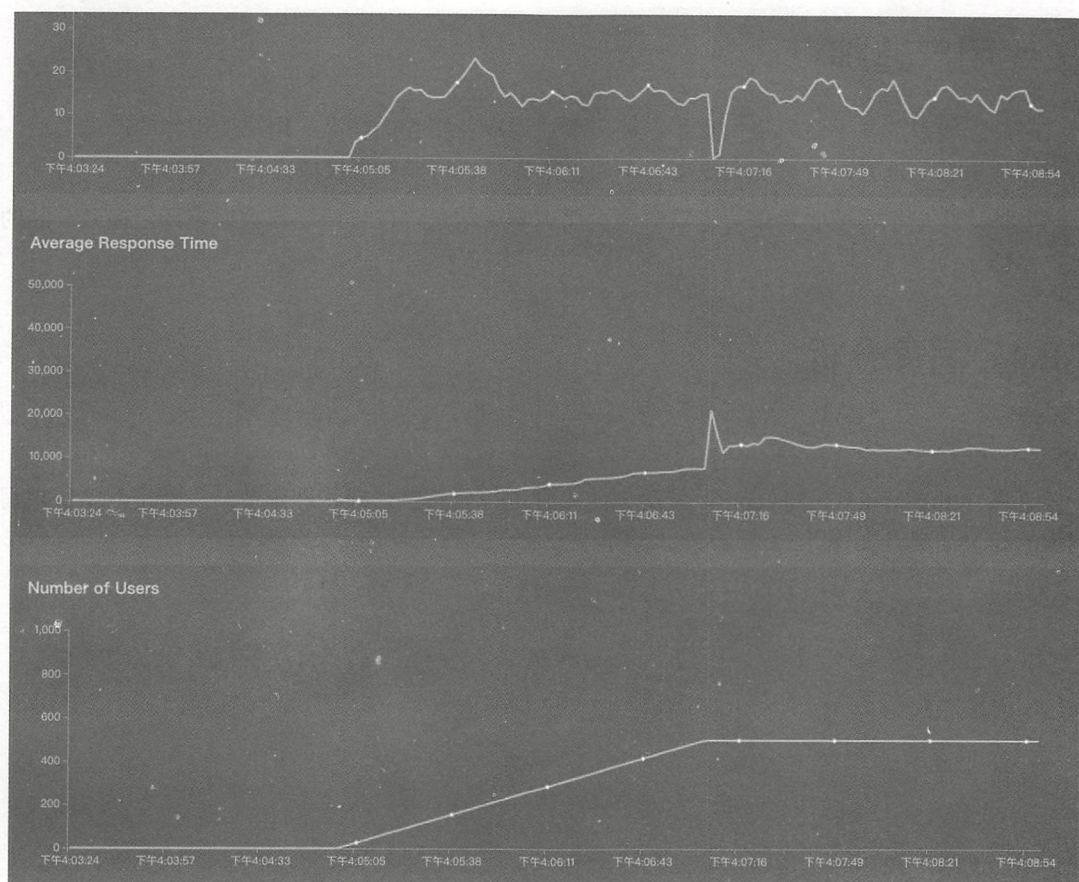


图 7-9

当用户到 500 的时候，平均响应时间为 13 秒，如图 7-10 所示。

400 个用户为 6s 左右的响应时间，如图 7-11 所示。

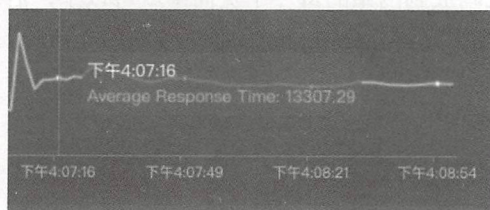


图 7-10

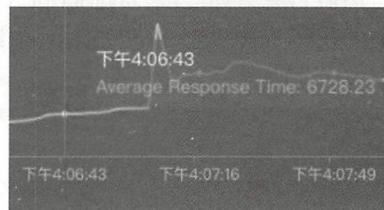


图 7-11

超过 8s，留存率会变成 40%，会让人感觉很慢，所以当用户数和用户产生的请求数同实际网站的运行情况差不多的时候，就需要搭建负载均衡来提高响应时间了。

当然这是在 dev 模式下的数据，而且没有使用 cache，在实际生产中，性能表现可能会更好。

7.2 用户追踪

产品上线后，并不是万事大吉了，还需要让用户使用你的产品。用户使用你的产品之后，遇到体验不好的地方，有可能会提交一个建议给你，也可能会直接放弃使用你的产品。

另外，我们也需要收集用户喜欢哪些页面内容等信息，以方便进行数据统计，提升产品的用户体验。

7.2.1 百度分析

1. 安装

进入 <https://tongji.baidu.com>，使用你的账号进行登录。选择管理菜单——新增网站，填写网站的相关信息，如图 7-12 所示。

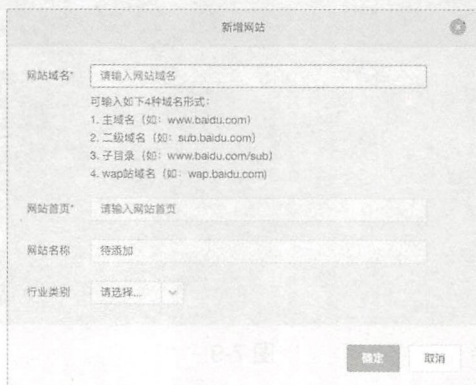


图 7-12

然后它会给你一段 JavaScript 代码，把这段代码放到网站的每一个页面中就可以了，后端渲染的代码直接放到布局模板中，前端渲染的代码直接放到 index.html 中即可，如图 7-13 所示。

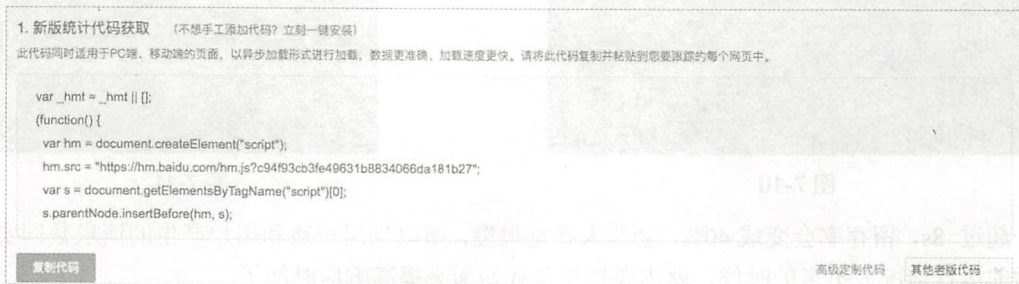


图 7-13



回到首页，当有用户访问你的页面的时候，百度分析就会开始收集用户访问数据了。你可能会看到类似图 7-14 所示的数据。

网站名称		浏览量(PV) ↓	访客数(UV)	IP数	跳出率	平均访问时长	操作
nodelover.me MiYogurt	今日	769	75	75	40.35%	00:07:43	
	昨日	2,666	194	190	41.03%	00:08:18	查看报告
	预计今日	2,303	197 ↑	201 ↑	--	--	

图 7-14

PV 代表 page view，表示页面浏览量；UV 代表 user view，表示有多少个人访问你的页面，基本上跟 IP 数差不多，单击“查看报告”链接，还可以看到如下分析图。

- 趋势图（UV、PV 的增长趋势图）；
- Top10 搜索词（用户在百度搜索何种词语找到你的网站）；
- Top10 来源网站（用户在哪个网站单击链接跳到你的网站）；
- Top10 入口页（第一次访问你的网站所在的页面）；
- Top10 受访页面（页面访问量排行榜）；
- 新老访客；
- 地域分布（IP 的地域分布）。

2. 原理

其实它的原理也是通过 JS 发送当前用户的访问信息，比如当前页面的 URL，可以通过 window.location.href 获取，来源可以通过 document.referrer 获取，等等。而且统计数据会发送 Ajax 请求到百度统计服务器，所以能获取 IP。收集好数据，进行可视化就实现了信息统计。其实我们也可以自己实现这样一个信息统计，不过比较麻烦，使用免费的百度统计即可，如果定制要求比较高，则还可以选择其他的付费服务。

3. 埋点

假如我们希望追踪用户单击了某个按钮怎么办？

```
_hmt.push(['_trackEvent', category, action, opt_label, opt_value]);
```

_hmt 是之前我们添加的 JS 代码得到的变量。_trackEvent 代表追踪时间。

- category 表示事件分类名称，比如视频，它代表一个分组；
- action 代表用户行为，比如单击；
- opt_label 记录的数据、视频名称；



- `opt_value` 记录的值、时间长度。

4. 单页应用

对于单页 App，由于 `_hmt` 只会在进入的时候执行一次，所以只会记录一次。为了解决这种问题，我们可以为每一个路由添加一个这样的钩子。

```
router.afterEach((to, from) => {  
  _hmt.push(['_trackPageview', to.fullPath]);  
})
```

7.2.2 Google 分析

Google 的分析具有实时的功能，但是访问有一些麻烦，所以建议优先使用百度分析。优先以修改 `hosts` 的方法来实现访问 Google: <https://github.com/googlehosts/hosts>。

1. 安装

访问 <https://analytics.google.com>，进行登录。单击左下角的“管理”按钮，选择“创建账号”，填写网站信息，像之前的百度统计一样，同样会给你一段代码，安装好即可。

2. 数据

Google 分析的优点是会有一个实时的活跃用户的区域。其他的地域分布、终端分布基本上跟百度分析一致，如图 7-15 所示。

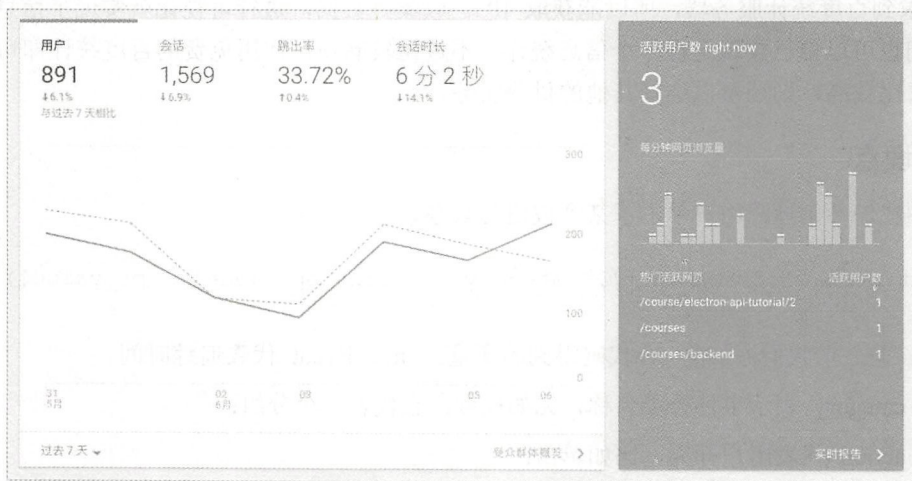


图 7-15



3. 更多高级功能

笔者也只使用了谷歌分析的一些基本功能，对于一些高级功能，读者可以在 <https://developers.google.com/analytics> 找到相应的文档。

在管理控制台上有一个发现按钮，里面也有一些资料可供参考。

相比较而言，百度分析 API 更加简单。

7.2.3 其他付费服务

- <https://www.growingio.com/>。
- <https://www.sensorsdata.cn/>。

7.3 前端性能分析与优化

不要把 CSS、Script 代码都写在 head 中，否则会阻塞渲染。

7.3.1 lighthouse

1. 安装 lighthouse

```
npm i -g lighthouse
```

2. 测试

```
lighthouse https://nodelover.me
```

会得到如图 7-16 所示的报告文档，这个站点笔者添加了比较多的特效与库，所以加载比较慢，第一次载入 CSS 代码花了 5 秒。

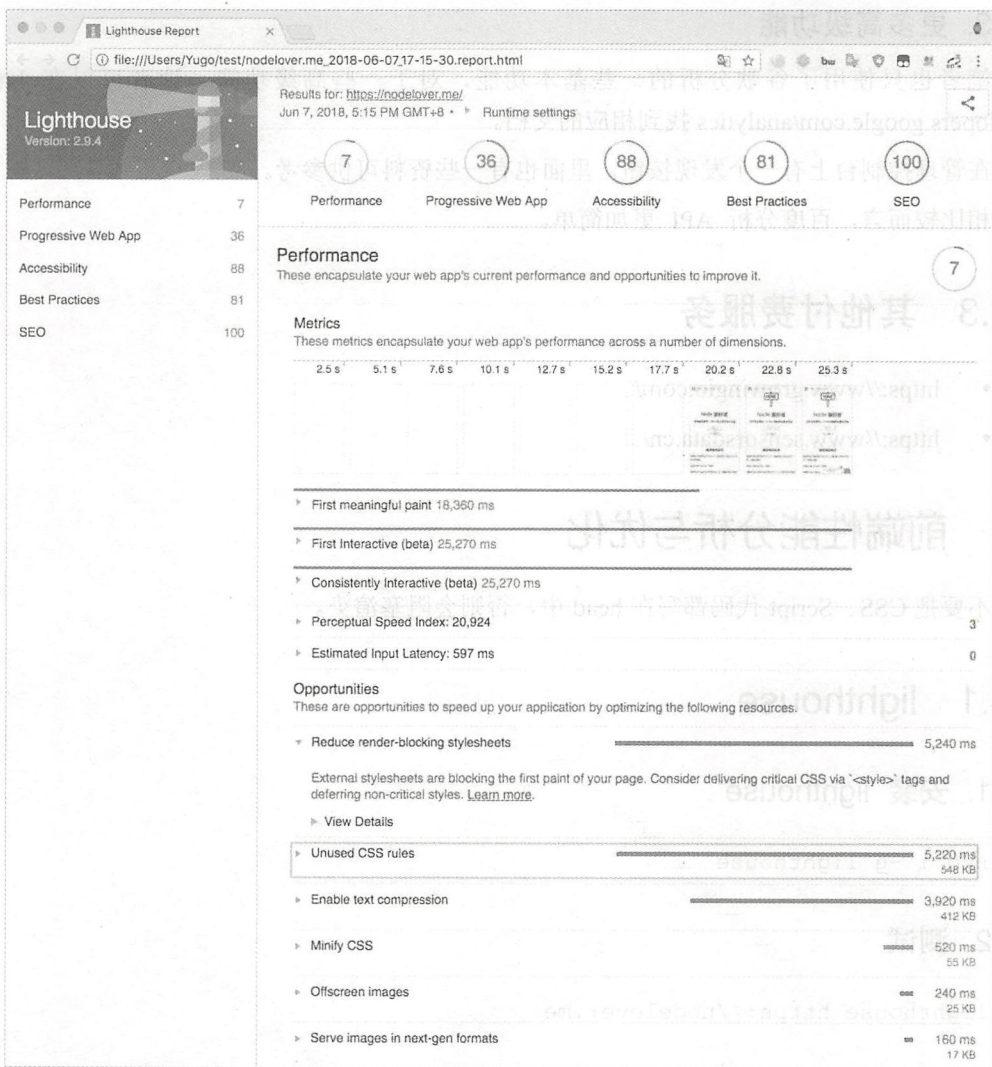


图 7-16

每一个标红的地方都会有一个 `lean more`，它会告诉你如何处理这个问题。

7.3.2 sonarwhal

1. 安装

```
npm install -g --engine-strict sonarwhal
```



2. 初始化

```
sonarwhal -init
```

3. 开始

```
sonarwhal -f excel https://nodelover.me
```

打开 Excel 文件，里面会有错误信息，然后根据 sonarwhal 的验证规则修改你的网站，如图 7-17 所示。

Rule id	Issue
axe	Error executing script: "TIMEOUT". Please try with another connector
disown-opener	'' is missing 'rel' values 'noopener', 'noreferrer'
highest-available-document	Meta tag is not needed
html-checker	A document must not include both a "meta" element with an "http-equiv" attribute whose value is "content-type", and a "meta" element with a "content-type" attribute
html-checker	A "meta" element with an "http-equiv" attribute whose value is "X-UA-Compatible" must have a "content" attribute with the value "IE=edge"
html-checker	Consider using the "h1" element as a top-level heading only (all "h1" elements are treated as top-level headings by many screen readers and other tools)
http-cache	The cache should not be cached, or have a small "max-age" value (180): max-age=604800
http-compression	Should be served compressed with Zopfli when gzip compression is requested.
http-compression	Should be served with the 'Vary' header containing 'Accept-Encoding' value.
http-compression	Should be served compressed with Brotli when Brotli compression is requested over HTTPS.
meta-charset-utf-8	A charset meta tag was already specified
no-disallowed-headers	'Server' header value contains more than the server name
no-friendly-error-pages	Response with status code 404 had less than 512 bytes
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/ans/2.2.0/ans.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/bulma/0.5.3/css/bulma.min.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/highlight.js/9.12.0/styles/atom-one-light.min.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/font-awesome/4.7.0/css/font-awesome.min.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/animate.css/3.5.2/animate.min.css
no-protocol-relative-urls	Protocol: relative URL found: //use.fontawesome.com/releases/v5.0.6/js/all.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/slick-carousel/1.8.1/slick.min.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/slick-carousel/1.8.1/slick-theme.min.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/sweetalert/1.1.3/sweetalert.min.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/plvr/2.0.16/plvr.css
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/plvr/2.0.16/plvr.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/ans/2.2.0/ans.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/ans/2.2.0/ans.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/ans/2.2.0/ans.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/sweetalert/1.1.3/sweetalert.min.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/Caret.js/0.3.1/jquery.caret.min.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/at.js/1.5.4/at.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/jquery-timexgo/1.5.4/jquery-timexgo.min.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/headroom/0.9.4/headroom.min.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/headroom/0.9.4/Query.headroom.min.js
no-protocol-relative-urls	Protocol: relative URL found: //cdn.bootcss.com/slick-carousel/1.8.1/slick.min.js

图 7-17

4. 在线版本

打开 <https://sonarwhal.com/scanner>，输入你的网址，笔者的站点扫出来的错误有点多，基本上都会给出解决方案，如图 7-18 所示。

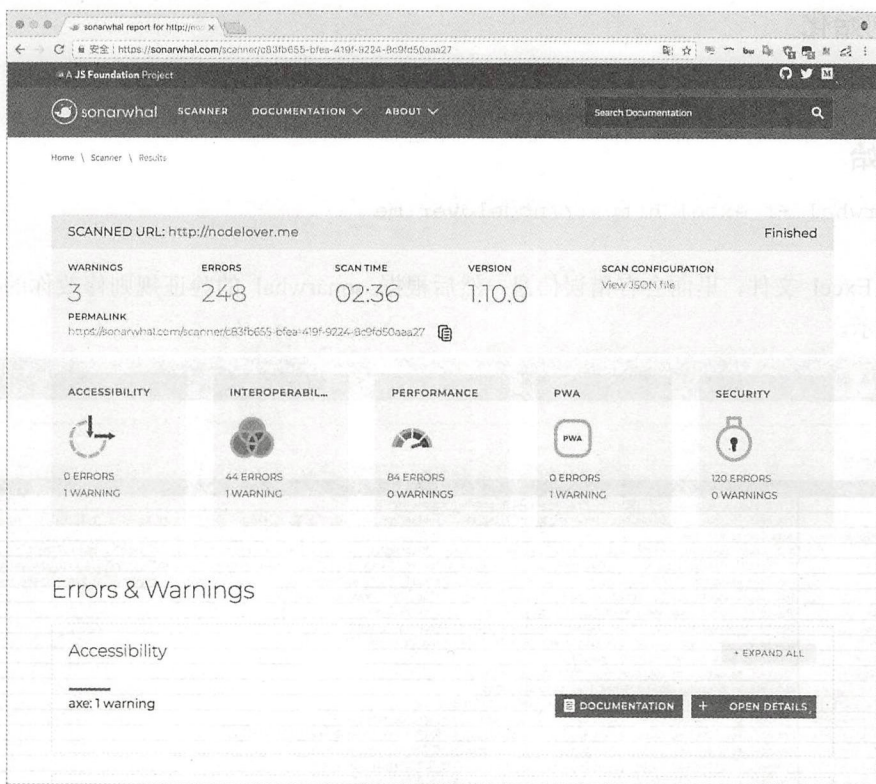


图 7-18

sonarwhal 和 lighthouse 都是参考而已，不是业界硬性标准。

7.3.3 图片压缩

用 webpack 打包过的 JS 文件和 CSS 文件基本上都已经压缩过了，所以这些文件不需要关心，我们需要关心的是静态图片，目前 webp 格式的图片其实比较小，使用 sharp 库可以非常方便地进行处理。

笔者对 sharp 进行了封装，做了一个桌面端的应用，只支持 Mac，地址为 <https://github.com/MiYogurt/mmmmh>，可以用来对图片进行压缩、转换。

也可以使用其他工具：

- Pngyu（支持 macOS、Windows）；
- ImageOptim（支持 macOS）；
- limitPNG（支持 Windows）。



推荐的方式还是使用 loader 自动处理，比如 image-webpack-loader 处理，同样支持处理成 webp 格式。

7.3.4 错误上报

有的时候代码总是会报错，但运行在用户的浏览器上，我们无法看到，所以必须要有一些工具把错误给记录下来。

1. 提供商

- <https://www.fundebug.com>;
- <https://www.bugsnap.com>;
- <https://sentry.io/welcome>.

第一个是国内的，以上都会提供一些 SDK 与免费的额度，可以非常方便地接入已有的系统。

2. 接入 Koa.js

```
app.on("error", fundebug.KoaErrorHandler);
```

接入 Egg.js，在 onerror 插件中处理即可。

3. 接入 Vue.js

```
Vue.config.errorHandler = function(err, vm, info)
{
  var componentName = formatComponentName(vm);
  var propsData = vm.$options && vm.$options.propsData;

  fundebug.notifyError(err,
  {
    metaData:
    {
      componentName: componentName,
      propsData: propsData,
      info: info
    }
  });
};
```

只需要几行就可以上报错误信息，各个服务提供商都有对应的 SDK，方法大同小异。



7.3.5 接收用户反馈

当然也可以创建公众号、留下联系邮件、单独的反馈页面，或者使用服务提供商。<https://leancloud.cn> 提供了一些免费的存储服务，用来做反馈组件足够了。

hotjar

进入 <https://www.hotjar.com>，注册你的账户，像百度统计一样，它会要求你添加一段 JavaScript 代码来进行安装。在后台可以定制显示的样式，有一定的免费额度，高级功能还支持用户埋点。

它支持选中页面的某个元素，然后提交用户反馈，如图 7-19 所示。

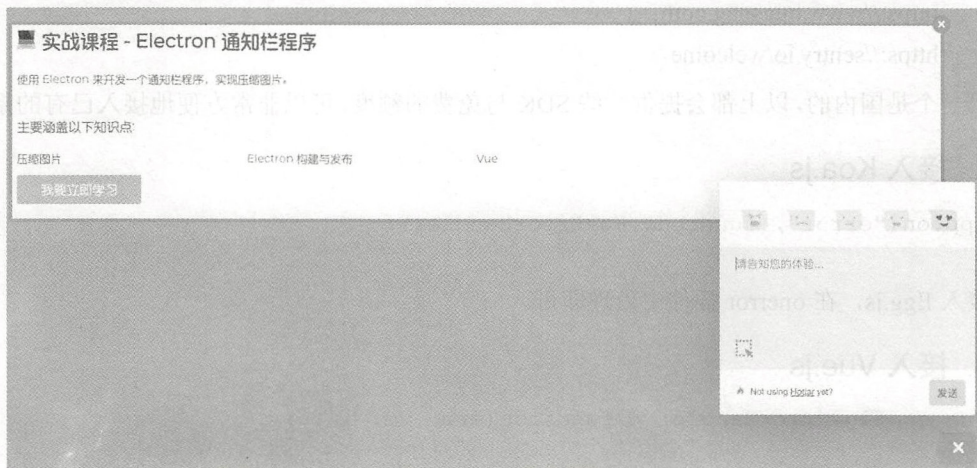


图 7-19

在线 IM 通信

这类基本上都要付费，没有免费的额度。

- <http://www.daovoice.io>;
- <https://www.intercom.com/pricing>;
- <http://www.udesk.cn>;
- <https://www.xiaoduoai.com>;
- <https://www.easemob.com>。

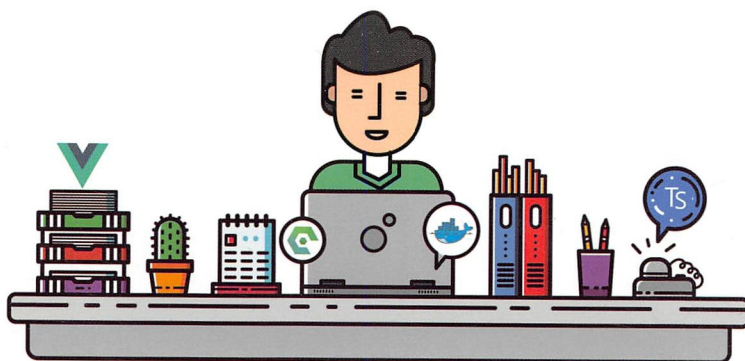
更多 feedback 工具可以查看 <https://mopinion.com/30-best-customer-feedback-tools-an-overview> 这篇文章。

好/书/分/享



拒绝堆砌臃肿,支持纯正原创

欢迎投稿: chenxm@phei.com.cn



本书以实现一个类似Dribbble的应用为例，将Node.js的技术点贯穿前后端的开发，整合Egg.js、Vue.js、Docker实现持续集成、持续部署的前后端分离应用。本书不局限于对Egg.js、Vue.js、Docker的讲解，书中还分享工作中必须要懂得的开发常识，比如如何对接服务（支付宝支付对接）、开放服务（通过OAuth开放API给第三方）。



博文视点Broadview



@博文视点Broadview



责任编辑：陈晓猛

封面设计：李 玲

上架建议：Node.js

ISBN 978-7-121-34912-6



9 787121 349126 >

定价：89.00元